Home

Contents

Welcome to the Hero Lab Authoring Kit Wiki

The Authoring Kit for Hero Lab provides a vast array of capabilities, and those capabilities will continue to evolve with the product. As such, we needed a means of documenting all those capabilities that could readily adapt and evolve as well. We concluded that the best way to accomplish this is to create a wiki that we can extend on an ongoing basis. As an added bonus, the wiki can also enable users to share tips and suggestions.

If you are not familiar with wikis, you can think of them as an intelligently structured assortment of web pages. For information on using this wiki, please refer to the User's Guide (http://meta.wikimedia.org/wiki/Help:Contents) .

Wikis are designed to be easily searched, so you can enter whatever term you are interested in and quickly find all the various entries pertaining to that topic. This will be invaluable as you become proficient with the Authoring Kit and want information on specific capabilities. Until you reach that point, you can simply start with this page and follow the various links below to read through all the various topics.

Introduction to the Kit

The goal of the Authoring Kit is to provide everything you need to create and/or edit data files for Hero Lab. When adding material to an existing game system, you can typically utilize the integrated Editor within Hero Lab. However, if you want to create data files for a new game system, you can use the information provided in the Authoring Kit to achieve that objective.

Hero Lab is the first tool that offers a versatile enough engine to handle all the complexities of virtually every RPG system. There is no practical way to develop an engine that inherently handles all this complexity without putting a substantial amount of power and flexibility in the hands of the data file author. With that power and flexibility comes a great deal of material to digest, though, which can make the Authoring Kit seem daunting at first.

To compensate for this, we've invested a great deal of time and effort to simplify and streamline everything, which extends well beyond just refining the engine and how everything works. The documentation is structured to introduce you to concepts in an incremental fashion and make it easy to find the information you need. We've provided a fully operational set of data files as a starting point for your own projects that can be readily adapted to any game system. We've even included a complete walk-through that details how to transform the starting data files into a full-fledged implementation of the Savage Worlds game system.

It's our sincere hope that you'll find the Authoring Kit reasonably straightforward to use and that we've made it possible for you to create quality data files for all your favorite games.

Documentation Conventions

For brevity, the Authoring Kit will often be referred to as simply the "Kit" and Hero Lab will typically be referred to as "HL".

Within the Kit documentation, there are a few conventions utilized. Anytime that important points arise within the text, they will be flagged appropriately using one of the techniques below.

WARNING!	Used to flag the most critical items, which have significant impact on the usability and maintainability of your data files

IMPORTANT!	Identifies considerations that impact the way in which you create your data files, but the results won't usually be horrible if you ignore them
NOTE!	Indicates topics that may influence your choices in data file creation in some situations

PDF Available

The Kit documentation is also available in PDF format for offline reference. The PDF version requires Adobe Acrobat Reader to view. You can download the PDF document via the link below.

http://www.wolflair.com/download/hp/hl_kit.pdf (last updated 23-Feb-2009)

A shorter version of the PDF, omitting the entries from Category: Authoring Examples, can be downloaded here:

http://www.wolflair.com/download/hp/hl_kit_noauthoring.pdf (last updated 23-Feb-2009)

IMPORTANT! The PDF version will usually **not** be as up-to-date as the online wiki, since the wiki is updated on an ongoing basis and the PDF version is updated only periodically. If there is a discrepancy between the PDF and the wiki, always treat the information here in the wiki as the most accurate.

Data File Authoring Topics

Each of the topics below will take you to detailed documentation on the corresponding facet of the Kit. A brief summary of each section is provided below as well. It is highly recommended that you start with the first topic in the list and work your way downward, just like you would normally read the chapters of a book in the sequence they appear within the book. Most chapters build upon the material from previous chapters, so skipping material may lead to some level confusion.

IMPORTANT! The Authoring Kit documentation is a work-in-progress and will continue to be expanded as the capabilities of the product continue to evolve. We've mapped out an extensive long-term plan for both the product and the documentation, and the general structure can be seen herein. However, many sections have not yet been written. These sections will appear as red links throughout the documentation and will be added over time to complete the documentation.

Basic Concepts and Terminology
This section covers all of the fundamental topics that the Authoring Kit is built upon. It's critical that you are familiar with all of these topics before continuing with the other sections.

Advanced Authoring Concepts
Building on the basic concepts, this section outlines more sophisticated mechanisms that you will likely want to leverage.

Kit Reference
Details regarding the syntax and structure for every facet of the Kit are spelled out in this section.

Authoring Examples
This section provides concrete examples showing how to add a wide range of features to your data files. This includes a walk-through that details creation of a complete set of data files for the Savage Worlds game system.

Techniques and Solutions [TBD]
Hero Lab offers a vast array of different capabilities, with different features being appropriate for different game systems. This section provides a laundry list of how to integrate the various mechanisms to tailor your data files to a particular game system.

Skinning the Interface [TBD]
The Kit provides you with the ability to completely change the visual look and feel of your data files. Once the basic functionality is in place, you can adapt the visuals however you like, just like has been done for the games Mutants & Masterminds and World of Darkness.

User Tips and Suggestions [TBD]
This section outlines an assortment of tips and suggestions that have been submitted by other users.

Legal Details

Contact Information

Company Website: www.wolflair.com (http://www.wolflair.com)
Technical Support Email: support@wolflair.com (mailto:support@wolflair.com)
Discussion Forum: support.wolflair.com (http://support.wolflair.com)

Category: TBD - Not Yet Written

Basic Concepts and Terminology

Context: HL Kit

This section identifies all of the fundamental concepts and terminology that form the backbone of the Kit. A brief summary is provided for general topic below. Click on a topic to get all of the details.

Contents

Glossary of Terms

There are numerous terms used throughout the Kit, and they are generally introduced in an incremental fashion. However, if you want one place as a central location for all terms, this is it.

Unique Ids

Just about everything within HL data files is assigned a unique id that serves to uniquely identify that object throughout the data files and enables it to be referenced by other objects. There are specific rules for unique ids that must be adhered to.

XML Files

All the data files you'll work with subscribe to the XML standard. A brief overview of XML files, as they pertain to the Kit, is provided in this section.

Structural Building Blocks

On a structural level, the Kit relies on an assortment of objects that serve as building blocks for everything pertinent to a given game system. These building blocks are outlined in this section.

Visual Building Blocks

Distinct from the structural building blocks for a game system, the Kit has a separate set of objects upon which the visual behaviors of the data files are built. The visual elements are outlined in this section.

The Physical Files

There are an assortment of different file types involved in a complete set of data files. They must reside in specific locations for HL to properly find and use them, and there are critical naming conventions that they must subscribe to.

Data Manipulation Basics

A significant part of data file creation is manipulating the various building blocks through scripts and other related mechanisms. This section details the basics to provide valuable context before delving into them indepth in the following sections.

Manipulation of Visual Elements

With the vast diversity of RPGs, the author needs to decide how information is presented to the user and how it responds to the user. The manipulation of the various visual elements is a critical element within any set of data files.

Data File Development Process

Now that you've been introduced to all the basic pieces, it's time to look at how everything comes together. The overall process of developing data files is outlined in this section.

Category: Basic Concepts and Terminology

Glossary of Terms

Context: HL Kit … Basic Concepts and Terminology

The Kit has an array of terminology that you'll need to become familiar with. Most of the fundamental terms are defined below. Other terms are defined as they are introduced within the documentation. The terms below are those that you will likely come into contact with in a variety of contexts during data file creation. Please familiarize yourself with all of these terms, as the rest of the Kit documentation assumes you are familiar with them.

This section provide merely a quickly summary of each term. You'll find more in-depth information on most of these terms in subsequent sections of this documentation.

unique id	Just about everything within the data files is assigned a *unique id* (often shortened to *id*) that serves to uniquely identify that object throughout the data files. Each object can then be referenced by other objects based on that unique id.
hero	A *hero* is the character that the user is creating and evolving within HL.
portfolio	A *portfolio* represents the collection of heroes being actively created by the user. In many cases, a user will only be creating a single character, but the portfolio can encompass dependent's, henchmen, allies, an entire adventuring party, etc.
thing	A *thing* encapsulates all of the characteristics of an object the end-user manipulates for a particular game system. These include classes, spells, weapons, skills, etc.
field	Each thing can have an assortment of values associated with it. Each of these values is a *field*, with some fields being fixed and others changing dynamically in response to user actions during hero creation.
component	All things can be grouped into sets of related behaviors (e.g. weapons, armor, spells, etc.). Each of these groupings is a *component*, with each component having a pre-defined set of fields that describe all instances of that component. Every thing is an instance one or more components.
component set	A *component set* represents a collection of one or more components. Each component corresponds to a particular aspect of the game system, and component sets allow those components to be blended into useful combinations. For example, a game system might have separate components for armor and cyberwear, but some cyberwear might also be armor, in which case a component set could be defined for cyberarmor that blends the two components into a single set.
pick	When a thing is added to a hero, an instance of that thing is created, and that instance is called a *pick* (because it will typically have been picked by the user). Since it is possible to add multiple instances of the same thing to a hero, and each instance can be configured differently, a separate term is needed for an added thing.
entity	Some things are complex and highly customizable by the user (e.g. vehicles, magic weapons, etc.). When that occurs, an *entity* is used to describe the customizations that can be performed within a separate level.
gizmo	Like picks, when an entity is added to a hero, an instance of that entity is created, and that instance is referred to as a *gizmo*. For example, multiple custom magic weapons can be added to the hero, each configured differently, with each being a distinct gizmo and all being based upon the same entity.
container	Both heroes and gizmos hold picks. When no distinction is needed between a hero and a gizmo, they are generically referred to as a *container*.
portal	Both on the screen and within character sheets, a *portal* represents a distinct visual mechanism for presenting the contents of a field or some other information to the user. Some portals will accept user input and others coordinate the presentation of large set of data through tables.
style	The visual look of a game system will be standardized so that all portals of a given type will share common characteristics like color, font, etc. To eliminate the need for re-defining this information for every portal, a number of *styles* are defined and every portal then refers to a style for its characteristics.
template	Portals are grouped into *templates* to describe how all of the information for a thing should be presented to the user. By defining labels and appropriate visual components to show the contents of fields, a template defines the way things are viewed by the user.
layout	When presenting information to the user, templates and individual portals will need to be combined, and this is achieved through *layouts*.
panel	On the screen, all information is organized for presentation to the user within *panels*, with a different tab panel corresponding to each tab across the top of the main window.
sheet	A *sheet* corresponds to a character sheet being output for the hero, usually to the printer.
table	Throughout the user-interface, the Kit uses *tables* to present a scrollable list of templates, wherein each item in the table presents the contents of a single thing or pick from a related set. Tables can sometimes support user-selection as well, in which case a separate selection table is used to choose from.

script	The Kit provides a simple programming language through which you can define *scripts* that are allow you to implement the nuances of every game system.
phase	Evaluation of scripts, rules, and other operations must be scheduled within the engine to occur in a controlled sequence. To this end, each game system defines a set of *phases* that break up the sequence into logical chunks.
priority	Within each phase, *priorities* are assigned that control the sequence within that phase.
eval script	Whenever the selection of a thing causes chain reactions to occur within the hero (e.g. a magic item that increases the wielder's strength), an *eval script* is written and attached that performs the necessary actions.
eval rule	Every game system has an assortment of constraints that govern hero construction, such as minimum and maximum limits, allowed combinations of abilities, etc. To enforce these constraints, *eval rules* allow these requirements to be validated.
task	Each eval script and eval rule is scheduled as an individual *task* within the HL engine for processing.
procedure	A *procedure* is a fragment of a script that is defined separately and can be re-used from within multiple different scripts. For example, if a game system has multiple pieces of equipment that trigger the same basic set of adjustments to the hero, a procedure could be used to put all the logic in one spot and re-use it.
tag	Each individual characteristic of a thing is identified via a *tag* associated with that thing. Each tag represents an individual facet of a thing, such as the level of a spell, whether a weapon is ranged or not, etc.
tag group	While individual tags identify a specific characteristic of a thing, a *tag group* identifies a collection of related tags. For example, spells in the d20 System game system each belong to a particular "spell level", and each spell could be assigned the tag "1", "2", "3", etc. from the tag group "spelllevel".
tag expression	Things will often have many different tags assigned to them. To identify the things that meet a select set of requirements, various tags will be tested for via a Boolean expression referred to as a *tag expression* (or *tagexpr* for short).
version	The term *version* refers to the version of a set of data files (or possibly the Hero Lab product). In order to keep track of updates to data files, a version number is assigned to the data files, which is used both by the end-user and by HL to track when changes are made.
element	Since the Kit uses XML for its underlying file structures, and the term *element* has an established meaning within XML, this term is used to reference objects that are implemented as XML elements within data files.
attribute	Similar to the use of the term element, the term *attribute* has an established meaning within XML, so this term is used to reference anything that is implemented as XML attributes within data files.
bootstrap	There are times when one thing or entity needs to automatically add additional things to the container, such as the special abilities conferred for a particular race. These automatically added things are considered to be *bootstrapped* by the object that adds them.
live	Both structural elements (e.g. picks and gizmos) and visual elements (e.g. panels and layouts) are considered to be either *live* or *non-live*, depending on whether they satisfy various requirements. When live, an object behaves normally. When non-live, an object behaves as if it does not exist (i.e. was never added to the container or is never shown to the user).
linkage	Logical associations between one thing and another can be established via *linkages*. This allows a script to be written for a component that is then inherited by all derived things, but with each individual thing defining a potentially different thing as its associated linkage.
dossier	An ordered collection of character sheets is referred to as a *dossier*.

Category: Basic Concepts and Terminology

Unique Ids

Most objects within HL are assigned a unique identifier so that they can be easily referenced throughout the data files. For example, each thing needs to have a unique value. HL uses unique ids for naming objects. Unique ids must comply with a number of rules, as outlined below. These rules enable HL to achieve significant performance optimizations at run-time, so these restrictions are important.

1. Unique ids can be a maximum of ten (10) characters in length.
2. Unique ids may only contain the standard alphabetic characters (A-Z and a-z), numeric characters (0-9), and the underscore ('_').
3. Unique ids are **case sensitive** ("foo" is not the same as "Foo" is not the same as "FOO").
4. While it is technically legal for a unique id to start with a numeric character (0-9), it is generally NOT a good idea to do so. If a unique id begins with a numeric character, it will not be able to be used in most scripts, rules, and tag expressions (since starting with a numeric character causes Hero Lab to think the unique id is a numeric value).
5. All unique ids must be unique within a defined context. For example, it is not valid to have two different tag groups with the unique id "mygroup". However, it IS perfectly reasonable to have a tag group named "mine" and a rule set that is also named "mine", since they represent different contexts.

NOTE! The notable exception to #4 above is for tags, where purely numeric ids are common (to represent ranges, levels, resource costs, etc.). Since tags are always referenced with their tag group, a purely numeric tag id incurs none of the liabilities mentioned above.

XML Files

Context: HL Kit … Basic Concepts and Terminology

Contents

Overview

All HL data files are stored as XML files, as are saved portfolios and most other files intended for user access. Consequently, you'll need to be familiar with the structure of these files in order to manipulate them appropriately. This section outlines the details you'll need to know for this purpose.

If you are not already familiar with XML, it is quite easy to learn, since XML uses simple text files that can be easily created or modified. For additional information on XML, there have been countless books published on the topic and there are extensive resources available on the internet. The official site can be found at the following link. http://www.w3.org/XML/

HL utilizes only the basic mechanisms of the XML standard, so HL files are quite simple to work with. Since XML and HTML both derive from the same set of standards, anyone even tacitly familiar with HTML will be able to pick up XML very readily – at least to the complexity level employed by Hero Lab (or the lack of complexity, actually).

There are numerous commercial, shareware, and freeware tools available for easily editing XML files. Each has advantages and disadvantages, so you will need to make the determination of which tool is "better" for yourself. It's also perfectly reasonable to edit XML files with a simple text editor, although you'll want an editor that at least offers line numbers, since errors are reported with line numbers to allow easy correction of problems.

Basic XML Terminology

You are assumed to be familiar with XML before attempting to write data files for HL. However, it's quite likely that you may be reviewing this documentation before deciding whether you want to try your hand at writing data files, in which case you might not know XML yet. So we've provided very basic definitions of a few fundamental XML terms below to help you better understand the Kit documentation.

Element	HL data files are comprised of XML elements that define all of the information for a particular game system.
Attribute	When creating data files, almost all information is conveyed through the use of attributes within the XML file format. Each XML element contains an assortment of zero or more attributes, where each attribute defines a specific piece of information about the element.
PCDATA	When a block of free-form text is required for an element within Hero Lab data files, that text is specified via the use of XML PCDATA. When using PCDATA, remember that you must enclose the entire text within a CDATA block as a wrapper if you utilize any of the XML reserved characters (see below).
CDATA	If you need to include special formatting and/or reserved XML characters within a PCDATA region, you will want to wrap your text within a CDATA block. A CDATA block simply prepends the text with the character sequence "<![CDATA[" and terminates the block with the sequence "]]>". The list of reserved XML characters is defined further below.
DTD	Every XML file must be assigned a formalized structure for its contents. The formal specification of an XML file's structure is via a DTD (short for Data Type Definition). An appropriate DTD should be included with this documentation for every public HL file format.

Reserved XML Characters

The XML language has a number of reserved characters that have special meaning. If you need to use these characters within your data files, an appropriate replacement must be used in accordance with the XML language specification. Alternately, a CDATA block can be used within a PCDATA region. These special characters are documented in detail within any reference on XML, but they are repeated below for convenience. Any time you need to use one of the characters below within HL files, use the corresponding character sequence given on the right.

<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;
literals	If you need to specify a character with a code of 128 or higher, you need to specify the character as a literal. The syntax for this is "&#ddd;", where "ddd" is the decimal value of the character code (e.g. "&#149;"). You can also specify hexadecimal values via the syntax "&#xdd;" (e.g. "&#xAE;").

XML Comments

If you are editing data files by hand, then you can freely insert comments into the XML data files using standard XML syntax. Comment blocks begin with the character sequence "<!—" and end with the sequence "—>". Any number of lines of text with any contents can appear within an XML comment block. Comments within XML files may **not** be nested.

For example, the following XML includes a comment that effectively omits the "dropped" element from the document, including all of its attributes and the "dropchild" child element as well.

```
<document>
  <first attrib="value"/>
  <second attrib="value" another="junk">
    <child>This is PCDATA</child>
<!--
  <dropped attr1="x" attr2="y" attr3="z">
    <dropchild attr="ignore"/>
    </dropped>
-->
  <third dummy="nothing">
  </document>
```

NOTE! If you create a data file outside of HL and then use HL's integrated Editor to edit the file, all comments will be thrown away by the Editor. When using XML comments, be sure to only edit those files within tools that will preserve the comments.

XML Character Encoding Sets

The XML specification identifies a number of character sets that can be utilized within a given document. Unfortunately, none of them fully support the Windows ANSI character set, and HL is a Windows application. HL assumes all XML documents subscribe to the XML character encoding set that most closely approximates Windows ANSI, and all characters within that set are assumed to be the corresponding Windows ANSI characters. This means that HL assumes all XML documents utilize the "ISO-8859-1" character set (more commonly referred to as Latin-1), with a number of exceptions that are detailed in the Kit Reference section of the documentation.

The identity element at the top of all XML files should specify an encoding of "ISO-8859-1" for completeness. If no encoding is given, ISO-8859-1 is assumed. An example is given below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

NOTE! There is an unofficial XML encoding named "Windows-1252" that properly reflects the Windows ANSI character set and is often used. However, various XML parsers do not recognize this encoding set due to its unofficial nature. In the interest of maximum compatibility, the modified Latin-1 set is used instead.

Category: Basic Concepts and Terminology

Structural Building Blocks

Context: HL Kit … Basic Concepts and Terminology

Overview

In order to support all RPG systems, HL has distilled out a collection of generalized building blocks that can be used to construct the mechanics of each game system. The engine leverages these flexible, fundamental building blocks in a highly object-oriented fashion. For example, every game system object (spell, feat, skill, class, etc.) has its distinct structure defined, and every individual object is a specific instance of the appropriate type. Each game system defines its own unique set of object types and all of the individual objects of each type. For example, the data files for the d20 System define object types for classes, feats, skills, spells, etc., while the data files for the World of Darkness game system define object types for clans, covenants, merits, disciplines, etc.

The data files for each game system dictate the set of object types available for that game system, as well as the set of objects for each of those types. The Kit provides an assortment of pre-defined building blocks that you can use, adapt, replace, and extend. The net result is an extensible framework that provides a substantial set of core functionality and allows significant customizability for authors to adapt to virtually any game system.

Topics

Each of the structural elements is outlined in the topics below. Further details on many of these elements can be found in the more advanced sections of the documentation.

- Heroes, Actors, and Portfolios
- Components and Component Sets
- Things
- Fields
- Tags and Tag Groups
- Picks
- Entities and Gizmos
- Containers
- Minions and Masters
- Leads
- Usage Pools

Category: Basic Concepts and Terminology

Heroes, Actors, and Portfolios

Context: HL Kit ... Basic Concepts and Terminology ... Structural Building Blocks

Heroes and Actors

The fundamental purpose of HL is to facilitate the creation and management of characters for role-playing game systems. Within HL, individual characters are typically referred to as "heroes", although each game system can define an alternate term for display to users if appropriate. Since the term "hero" can be renamed, and since "hero" isn't really a suitable term when creating villains, monsters, and other types of NPCs, the Kit often uses the term "actor" instead.

There is no difference between the terms "hero" and "actor" within the Kit. They are interchangeable.

Portfolios

HL saves actors within a "portfolio". Multiple actors may be saved within the same portfolio.

Each portfolio is self-contained and makes no references to any external files. If an actor has a character portrait image file associated with it, the image is thereafter managed and stored within the actor and portfolio, with no further connection to the original file.

Category: Basic Concepts and Terminology

Components and Component Sets

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Each distinct type of game system object is defined via a unique component set (often shortened to "compset"). The component set defines a specific set of characteristics for an object type. For example, in the d20 System, skills have a particular set of behaviors (e.g. the number of skill points assigned, the linked attribute, etc.), while weapons have a different set of behaviors (e.g. damage type, damage value, critical score, etc.). A separate compset is therefore defined for each type of object you want to create.

Compsets get their name because they are actually a set of one or more components. Various object types will have similar behaviors to each other. For example, all types of equipment have a cost and weight, whether the equipment is a weapon, armor, magical rope, or merely a wineskin. All of the details associated with the cost and weight of equipment should ideally be handled in a simple, consistent fashion. By defining a single component that handles everything to do with cost and weight, and re-using that within the various compsets for each equipment type, the Kit becomes powerful, flexible, and efficient for the data file author.

Through the intelligent design of components and component sets, you can re-use common logic across multiple similar object types. The Kit provides a number of standard components and compsets that you can readily build upon and/or adapt to the specifics of your game system.

Category: Basic Concepts and Terminology

Things

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Each specific game system object is always based upon a specific compset. These objects are referred to generically as "things" within Hero Lab. For example, the "Longsword" thing might be based upon the "Melee" compset, while the "Fireball" spell would be based on the "Spell" compset.

The compset dictates the general behaviors of all individual things that are based upon it. However, each thing then tailors its own nature and behavior, as appropriate.

All things possess a variety of basic characteristics that are inherently unique to each thing, such as a name and description. Thing also have various characteristics that are inherited from the components from which the thing is derived (which are dictated by the compset). Lastly, things have an assortment of special characteristics that can be optionally specified to customize their individual behavior, including scripts, rules, and a host of other facets (all described elsewhere in this documentation).

Some of you out there in reader-land might be wondering why we used the term "thing" instead of the more standard term "object". The reason is that there are other facets of the underlying HL engine and the Kit that are also very object-oriented in nature. We wanted to be able to use the term "object" in the documentation to refer to general object-oriented behaviors, without specifically linking the word to this one situation. Using the term "thing" for compset-derived objects gives us that ability.

Category: Basic Concepts and Terminology

Fields

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Overview

Field represent values that can be assigned to things. For example, a weapon might have a field to specify the damage it does, while a skill would have a field to track how proficient the character is with that skill. Fields can be specified as being either numeric or text-based in nature, with text-based fields requiring that a maximum character length be designated.

Fields are defined within components. As such, a given field is inherited by every compset that incorporates the component it is defined within. This means that the fields possessed by a thing are dictated by the compset it is based upon, with all the fields defined within the compset's constituent components being possessed by the thing.

Every component can define zero of more fields. No fields are actually required for a component, and there are times when using no fields is appropriate, but most components will be defined with at least one field.

Field Values

Fields can be assigned a default value when defined within a component. This value will be inherited as the default value for every thing that includes the field. When defining a thing, you can either inherit the default value for a field or specify the appropriate value to use for it.

The values of fields can be dynamically modified at run-time in response to events triggered by user selections. Many fields will be wholly derived at run-time, such as a character's attack skill with a particular weapon, which must be calculated from all the various factors that contribute to the final result. This is achieved via the use of scripts.

The Kit also supports the definition of fields as both arrays and matrices. If this is done, the maximum dimensions of the array or matrix must be specified and HL will allocate storage for all elements in the array/matrix. Further details on arrays and matrices are defined as an advanced concept.

NOTE! Any attempt to assign a text value longer than the defined maximum to a given field will result in the text being truncated at the maximum length.

Category: Basic Concepts and Terminology

Tags and Tag Groups

Context: HL Kit ... Basic Concepts and Terminology ... Structural Building Blocks

Contents

Overview

At the heart of HL's engine is an important concept that is used as one of the basic building blocks for data files: tags. Things can be assigned a virtually limitless variety of tags, where each tag represents a label that describes some facet of the thing. Tags are managed in sets referred to as tag groups. All of the tags in a given tag group are intended to have a related meaning, such as "race", "level", "color", etc. The set of tags and tag groups is unique to each game system.

Things can have any number of tags assigned from any number of tag groups. Sometimes you'll use zero tags and sometimes you'll use a dozen or more for a complex set of behaviors. You can also freely re-use tags on different types of things, and there are times when using the same tags on different types of things can be extremely powerful. One key advantage of tags is that they are fast and flexible when compared to fields. Modifying a field requires modifying the underlying component and then adjusting all of the things derived from that component, whereas tags can be added, modified, or deleted at any time. The most powerful advantage of tags, however, is that tags can be easily utilized in tag expressions, which are a powerful and flexible mechanism used extensively within data files (and discussed in detail elsewhere).

Referencing Tags

Since tags always belong to a given tag group, an individual tag is always identified with the tag group to which it belongs. This is critical, because the same tag could be used in two or more different groups, with each having a different meaning. For example, the tag "Common" could have two very different meanings in two different tag groups for some game systems. Within the "Language" tag group, "Common" might refer to the "Common tongue" used in the game world. However, within the "Rarity" tag group, the "Common" tag might indicate that an item is commonly available for purchase instead of being uncommon or rare.

When referring to tags, the notation used is called a tag template. This notation specifies the unique id of the tag group, followed by the '.' character, followed by the unique id of the tag. For example, the reference "Language.Common" would refer to the "Common" tag within the "Language" tag group. Remember this syntax, since tags are used extensively and you'll be seeing the tag template notation frequently.

REMINDER! Unique ids for almost all elements must be globally unique. The key exception is with tags, for which the unique id must be unique only within the containing tag group. This is possible because tags are always referenced with the associated tag group, so uniqueness is always ensured, even though two tags in different groups may have the same id.

Using Tags

In most cases, you'll be assigning a single tag from a given tag group to a thing. However, sometimes you'll find that the lack of a tag from a particular group can have important meaning. At other times, you'll find that you need to assign multiple tags from a given tag group to a particular thing. For example, spells in the d20 System have different casting behaviors (e.g. verbal, somatic, and material), with any number applying to a particular spell. In this case, you would assign the appropriate tags to a given spell, with the presence of a tag indicating the behavior and the absence of the tags indicating the opposite.

Dynamic Tags

In most cases, each tag group will explicitly define the complete set of tags that exist for that tag group. However, there will be times when a tag group needs to be open-ended for expansion. For example, consider the the "Language" tag group mentioned above. You could populate the group with all of the languages in the game system, but you can be sure some designer will come along at some point in the future with a new language for some new race. To handle these situations, a tag group can be designated as dynamic, allowing new tags to be defined on-the-fly as they are needed. When you define a new dynamic tag, you must specify all its various details.

Tag Values

Tags will usually be used to identify a non-value attribute of a thing. Once in awhile, though, tags will be used to convey a value in addition to an attribute. For example, each spell within the d20 System has a spell class and a level for that class (e.g. Wizard level 4, Cleric level 2). Since the same spell can apply to different classes at different levels, it would be difficult to track all that information easily. So one solution would be to define a "SpellLevel" tag group with tags that combine the class and the level, such as "Wiz4" or "Clr2". The data files could then identify a spell's class from the prefix on the tag and also get the value from the suffix. The tag value of a given tag is the determined by extracting the trailing digits from the tag's unique id and converting it to an integer value,. So the tag id "xyz123" would have the value "123", while the tag "a1b2c3d4" would have the value "4" (the extraction stops at the first non-digit encountered, starting at the end and working towards the front). If a tag id does not end in any digits (e.g. "black"), the tag's value is always treated as zero.

Tags Are Often Better Than Fields

The concept behind fields will likely be familiar to you, while the concept behind tags may not. This makes it quite possible that you will be inclined to use fields to solve most problems. However, you will benefit from learning how to leverage tags, since there are many situations where tags are a better solution than the use of fields.

Tags provide a fast and efficient means of assigning an attribute to something. Tags are ideal when valid values consist of a pre-defined set. For example, spells in the d20 system have a level, a range, an area of effect, and some combination of three different casting behaviors: verbal, somatic, and material. The range and area of effect have a very wide range of possible values, so a field is probably the best way to manage that characteristic. However, the spell level and casting behavior are ideal situations for a tag. There is a fixed number of spell levels and there are only three casting behaviors. Two tag groups could be easily defined to solve this (one for spell level and one for behavior), and individual tags could be defined for each level and each behavior. Appropriate tags would then be assigned to each spell corresponding to the spell level and casting behaviors.

A field could also be used for each of these, but tags are a better solution. This is especially true for the casting behavior, where using a field would either entail having a numeric code that is not immediately obvious (e.g. 1=verbal, 2=somatic, etc.), or it would require using a text value (e.g. "verbal", "somatic"). Using a text value would be prone to data entry typos over the span of hundreds of spells. In contrast, the use of the tag "verbal" makes the value obvious and the requirement that all tags match an existing set eliminates the possibility of typos.

Category: Basic Concepts and Terminology

Picks

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Consider a situation where an actor possesses two separate longswords. Both of these weapons correspond to the exact same thing that has been defined within the data files. Now consider what happens when a magical effect is applied to one of the longswords, but not the other. Similarly, consider a customizable weapon, such as a modern assault rifle, where one rifle might be configured with a laser scope and infrared optics, and a second rifle might be given recoil compensation. In each case, both weapons are based upon the same underlying thing, but the weapons must be kept distinct from one another and modified independently.

To handle situations like this, whenever a thing is added to an actor, it becomes a completely independent object. This new object is referred to as a "pick". Each pick references the original thing and inherits all of its characteristics as its initial state. However, all dynamic changes are applied independently to each pick.

IMPORTANT! Throughout the Kit documentation, the terms "thing" and "pick" will be used extensively. It is critical that the distinction between the two be clearly understood. Otherwise, the Kit documentation could become confusing, since every behavior that can be applied to things can also be applied to picks (i.e. picks are essentially a superset of things).

Category: Basic Concepts and Terminology

Entities and Gizmos

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Things and picks work for most situations, but they just aren't sufficient when you need to support something complex, such as a customizable vehicle or magic item. In situations like this, you'll need to define something that can itself contain different picks, thereby allowing for customization by the user.

The Kit utilizes an "entity" for this purpose. Each entity can be specified as starting out containing various things and/or tags, thereby allowing you to pre-configure various facets of the entity.

An entity is always attached as a child of a thing. A given thing can only attach a single entity, and a thing assigned a child entity will always attach that entity whenever it is added to an actor.

When a pick is added to an actor and that pick has a child entity, the new entity is referred to as a "gizmo". The distinction between entity and gizmo parallels the distinction between a thing and a pick. The entity is the definition, but the gizmo is the actual instance that has been added to the actor. Since it is possible to attach multiple instances of the same thing to an actor, and that thing can have a child entity, it's also possible to have multiple instances of the same child entity. Each of those is a distinct gizmo.

When added to an actor, each gizmo inherits its starting state based on the entity. If the entity is assigned various tags, those tags start out on the gizmo. Similarly, if the entity is assigned starting things, each of those things is added to the gizmo as a pick within that gizmo.

Category: Basic Concepts and Terminology

Containers

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

The term "container" is used to generically refer to both actors and gizmos, since both actors and gizmos contain an assortment of picks. Actors involve a great deal more behaviors than gizmos, and gizmos are **not** a proper subset of actors. However, whenever something applies equally to both actors and gizmos, it will be described as applying to containers.

Category: Basic Concepts and Terminology

Minions and Masters

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Many game systems involve the creation of actors that are secondary to a character. For example, hirelings and henchmen are a common situation. In the d20 System, wizard familiars and druidic animal companions are tied to the character in a secondary relationship. In Mutants & Masterminds, sidekicks and minions are purchased out of the character's available power points.

To support these various situations, the Kit provides a generalized mechanism to create and maintain a hierarchy of actors. This hierarchy consists of "masters" and "minions", wherein an actor is the master of any actors beneath it in the hierarchy and a minion of any actor above it.

In the same way that an entity can be defined and associated with a thing, so can minions. The key difference is that minions are always actors, and HL already knows how to create an actor, so there is no need to define a separate "minion" element. All that is needed is to associate a minion with a thing and specify any special characteristics for the minion. Whenever that thing is added to a character as a pick, a new minion is automatically created and customized appropriately.

The actor hierarchy does not have a maximum depth. This is necessary to handle situations where a character has a minion, and that minion has its own minion, etc.

Category: Basic Concepts and Terminology

Leads

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Not all actors are created equal. As such, their relationship to one another is critical in determining a few subtle, yet important, distinctions between actors. In a nutshell, a "lead" is any actor that is not a minion of any other actor (i.e. top-level).

Whenever a new portfolio is created by the user, an initial character is always created as well. This new actor is considered to be a "lead" actor. Similarly, whenever the user creates a new actor in an existing portfolio by means of the "Create New Hero" option on the "Portfolio" menu, the new actor is considered to be a "lead". And the same applies to any new actor that is directly imported from another portfolio, including a stock portfolio.

The exact implications of leads versus non-leads are described in the relevant sections elsewhere in this documentation. However, the distinction is important, so it is set forth here.

Category: Basic Concepts and Terminology

Usage Pools

Context: HL Kit … Basic Concepts and Terminology … Structural Building Blocks

Most of the time, the information managed for a character simply involves tracking the current state. However, there are times when a history of changes needs to be maintained. To handle these situations, the Kit provides a mechanism referred to as a "usage pool".

Usage pools track both the current net value for the pool and a history of adjustments applied to the pool, up to a configurable maximum history size. A usage pool can be defined for an actor or for an individual pick. This makes it possible to use a usage pool to track the overall experience and cash accrued for an actor, as well as for tracking the individual experience and cash accrued for a single journal entry.

Usage pools are also invaluable for tracking damage and similar resources. Since a history is maintained, it becomes possible to easily undo damage that was sustained. It's also possible to show the history to the user for review, such as the sequence of damage and healing that took place during a lengthy combat.

Category: Basic Concepts and Terminology

Visual Building Blocks

Context: HL Kit … Basic Concepts and Terminology

Overview

The visual presentation of each game system is wholly distinct from the underlying mechanics of the game system. However, it is no less critical to the overall usability of the data files. All the visuals are constructed from an assortment of generalized building blocks that are designed to allow you to tailor the interface to the nuances of each particular game system.

The overall structure of the visual interface is dictated by the way HL is designed. For example, the use of edit panels, summary panels, and tables of information will be common across every game system. Within that overall structure, though, the data file author can tailor the interface in a wide range of different ways.

In general, you will find yourself designing the contents of panels and sheets, which are very similar in how they work. Both contain one or more templates and/or layouts, which allow you to position groups of individual visual elements. Templates contain one or more portals, while layouts contain a combination of portals and/or templates. This general structure applies to all visual elements that you'll create as an author.

The Kit provides a variety of pre-defined, visual building blocks that you can use, adapt, replace, and extend. This should make it possible to get something working in relatively short order, while also allowing you to extensively customize the interface to the needs of virtually any game system.

Topics

Each of the visual elements is outlined in the topics below. Further details on many of these elements can be found in the more advanced sections of the documentation.

- Portals
- Templates
- Layouts
- Panels and Forms
- Sheets
- Scenes
- Dossiers
- Resources
- Styles
- Sort Sets
- Encoded Text

Re-Using Visual Elements

Visual elements are all globally defined. Consequently, you can define a template once and re-use it in multiple tables or layouts. Similarly, you can define a layout once and re-use it in multiple panels or sheets.

However, what if you want to display multiple pieces of information that share the same template from within the same layout? Or what if you want to re-use the same layout within the same panel or sheet? The Kit uses template references and layout references when specifying the use of these visual elements. Multiple references can re-use the exact same template or layout, with each reference being assigned a different logical name for use within the containing visual element. The logical name is then used in any positioning scripts for the containing visual element to uniquely identify the correct template or layout.

Category: Basic Concepts and Terminology

Portals

Context: HL Kit ... Basic Concepts and Terminology ... Visual Building Blocks

Contents

Overview

Portals are the finest level of control within a visual presentation. They provide access to individual fields within things and picks. Fields are hooked up to portals, thereby allowing the user to view and/or modify the contents of those fields. For example, a "label" portal could be associated with the "damage" field for a weapon, thereby allowing the contents of the field to be displayed to the user via that portal. If an "edit" portal were used instead, then the contents of the field could potentially be modified by the user.

Each of the different types of portals is summarized in the sections below. Complete details on each portal type will be found within the Kit Reference section of this documentation.

IMPORTANT! Portals are unlike most other elements used within the Kit. For portals that are defined within templates, the unique ids of those portals must only be unique within the context of the template. This means that you may freely re-use the same unique id for portals within different templates. For example, you could have a "name" or "delete" portal within multiple templates. Since portals within templates are only accessible through the template in which they are defined, their scope is uniquely restricted to the template, which makes the re-use of ids possible. Please note that this ability to re-use unique ids **only** applies to portals defined within a layout, so it does **not** apply to tables and other portals used directly within layouts.

Label Portals

Labels are the simplest type of portal and allow the display of information to the user. There are different types of label portals, depending on what the label contains and how it is to be displayed to the user.

Literal	Literal label portals display a fixed string of text.
Field-Based	Field-based label portals show the contents of a specific field.
Script-Based	Script-based label portals synthesize their contents via a script, with the results being displayed.
Titles	Title label portals present their contents with special formatting for use as a title.

Image Portals

Image portals enable the display of images to the user. There are different types of image portals, depending on what the portal contains and how it is to be presented.

Literal	Literal image portals always display the same, fixed image.
Field-Based	Field-based image portals show the image dictated by the contents of a specific field.
User	User image portals present the image that has been selected by a user.
Reference	Reference image portals allow a user-added image to be dynamically adapted for use at run-time (such as using a

tiny version of the first character portrait on the Tactical Console).

Edit Portals

Edit portals enable the user to edit the contents of a specific field. There are different types of edit portals, depending on what the portal contains and how it is to be presented.

Simple	Simple edit portals provide direct editing of a numeric or text field.
Date	Date edit portals offer structured editing that ensures the contents of the field always subscribe to specific rules (e.g. four digits for year, two digits for month, etc.).

Incrementer Portals

Incrementer portals provide the ability to edit the contents of a specific numeric field. An incrementer includes visual elements like a "+" and "-" button that allow the user to easily increase and decrease the value shown within the portal. Incrementers are typically used for attribute values, damage tracking, charges, and other similar mechanisms.

Checkbox Portals

Checkbox portals allow the user to toggle the state of a specific field between values of zero and one. The user sees the portal as an either/or toggle state.

Menu Portals

Menu portals enable the user to select one choice from a list of available choices. There are different types of menu portals, depending on what the portal contains.

Literal	Literal menus present a choice from a set of options that are pre-defined and fixed. They are suited for situations like the gender of a character and the method used for selecting ability scores in the d20 System.
Array	Array-based menus use a script or other technique to synthesize the list of valid choices the user can select from.
Things	Thing-based menus dynamically identify a group of things from which the user can select. They are useful when choosing a weapon type or spell type to assign a skill bonus.
Picks	Pick-based menus dynamically identify a group of picks from which the user can select. They are ideal when you need apply an adjustment to something that is restricted to what the character possesses, such as choosing a weapon that the character has.

Chooser Portals

Chooser portals allow the user to select a thing from a table of options that include full descriptions. Once the thing is selected, it is added to the character as a pick and becomes an integral part of the character. Chooser portals are perfect for selecting a race or an alignment, since exactly one selection must be made for the character, and the user will want to be able to review the details associated with each possible choice.

Action Portals

Action portals are presented as buttons that trigger a specific action when interacted with by the user. There are many types of action portals, depending on what action is desired.

Delete	Triggers the deletion of a pick from the table that added it.
Info	When clicked or the mouse is moved over the portal, full details on the pick are displayed.
Edit	Triggers editing of the gizmo associated with a pick by bringing up a suitable form for the purpose.
Form	Triggers the display of a specific tab panel as a form, such as is used to directly apply damage to actors from within the Tactical Console.
Trigger	Invokes a script, such as when applying damage to an actor or resetting the accrued damage.
Notes	Displays a form where arbitrary notes can be edited for a pick.
Load	Switches to the actor associated with the specified pick, such as is used on the Dashboard and Tactical Console.
Lock	Transitions the character into a locked state for the purposes of advancement.

Unlock	Transitions the character into an unlocked state for the purposes of advancement.
Master	Switches to the actor that is the "master" of the current actor.
Minion	Switches to an actor that is the "minion" of the current actor.
Manage Gear	Displays a menu through which the user can move gear between different containers.
Get Gear	Displays a menu through which the user can move gear between actors.
Start Combat	Begins a new combat within the Tactical Console.
End Combat	End the current combat within the Tactical Console.
New Turn	Starts a new combat turn within the Tactical Console.
Initiative Change	Incorporates any direct initiative changes made by the user within the Tactical Console.
Integrate	Integrates any pending actors into the current combat.
Sort Dashboard	Triggers a re-sort of the actors shown on the Dashboard.

Region Portals

Region portals make it possible to place a border around a rectangular region that doesn't correspond to a single portal. For example, if you want to visual group a few portals, you can define a region that encompasses all of them and place a border around the region, thereby placing a box around the group of portals.

Separator Portals

Separator portals allow you to put either a vertical or horizontal separator bar between sections of the display. Separators are an excellent way to visual group portals and make the interface more intuitive for the user.

Table Portals

Table portals present a collection of related picks in a tabular list. Each item in the table is displayed via a template that is specified with the table. The template specifies how to position the various facets of the pick within the table entry. There are different types of table portals, depending on what the portal contains and how it is to be presented.

Fixed	Fixed table portals display a table of picks that provides no means for the user to add or delete items.
Dynamic	Dynamic table portals display a table of picks that includes an option at the bottom to add new items to the table. If the "add" option is used, the user is presented with a list of things to choose from and the one selected is added to the container as a new pick.
Auto	Auto table portals display a table of picks that includes an option at the bottom to add new items to the table. If the "add" option is used, a specific thing is added to the container as a new pick, which the user can subsequently customize. The journal is an example of an "auto" table, where adding a new item automatically adds a new journal entry.

Special Portals

Special portals are those portals that have highly specialized uses and cannot generally be customized by the author in term of behavior. However, they do need to be properly placed within the interface, so a special portal is used to represent the mechanism and allow placement. There are a number of special portals, as outlined below.

Edit Settings	Displayed as button that allows the user to edit configuration settings when clicked. This portal is typically used on the Configure Hero form.
Settings Summary	Displayed as a grid that shows the currently selected configuration settings. This portal is typically used on the Configure Hero form.
Alliance	Displayed as a menu that allows the user to select whether the character is to be considered an ally or an enemy for the purposes of the Tactical Console. This portal is typically used on the Configure Hero form.

Output Portals

The output of character sheets entails a number of critical differences from the interactive visual elements used on-screen. As such, a different set of portals is used to render character sheet output. The presentation behaviors of output portals is similar, yet distinct, from interactive portals, and the set of available output portals consists of those outlined below.

Label	Renders text onto the character sheet, with the text being driven by any of the different mechanisms supported by on-screen label portals.
Image	Draws an image onto the character sheet, with the image being handled in any of the different ways supported by on-screen image portals.
Table	Renders a table of picks to the character sheet, using a template to specify the contents of each item within the table.

Category: Basic Concepts and Terminology

Templates

Context: HL Kit ... Basic Concepts and Terminology ... Visual Building Blocks

Templates contain one or more portals and represent a rectangular region within the containing layout. The template is responsible for coordinating the position of its own portals within its boundaries.

Every template is associated with a specific thing or pick. All of the portals within the template are associated with that thing or pick. Therefore, all fields associated with portals reference that field with the thing/pick associated with the template. For example, if a template is associated with the "pistol" pick and the template contained a portal associated with the "range" field, that portal would display the contents of the "range" field of the "pistol" pick.

IMPORTANT! Since templates are associated with a thing or pick, templates may **not** contain portals that are precluded from having field associations. This means that table and chooser portals may not be used within templates.

Category: Basic Concepts and Terminology

Layouts

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

Layouts contain one or more visual elements, where those visual elements can be templates and certain types of portals. Layouts also represent a rectangular region within the containing panel or sheet. The layout is responsible for coordinating the position of its contained visual elements within its boundaries.

Layouts make it easy to group related visual elements together and position them as a atomic unit. For example, within the d20 System data files, there is a layout that manages the selection of class-specific special abilities, another layout for viewing the special abilities for the class, etc. By using layouts, new edit panels for custom classes can be quickly constructed by combining the appropriate layouts for the separate features of a given class.

IMPORTANT! Since layouts are not associated with a thing or pick, layouts may only contain portals that do not have field associations. This means that, in general, only label portals, image portals, chooser portals, and table portals may be used within layouts. In the case of labels and images, the field-based versions of those portal types may not be used within layouts, since no field association exists.

Category: Basic Concepts and Terminology

Panels and Forms

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

Panels and forms are the top-level visual element used on-screen. Panels define the contents of pre-defined regions within HL. This includes the various tab-based panels used within the interface, as well as the summary panels shown on the right. Forms behave very similarly to panels and are used to define the contents of standalone windows that HL manages. These include the Configure Hero form, the Tactical Console, and forms for editing the contents of gizmos (e.g. custom magic weapons within the d20 System).

Panels and forms have a few important behavioral differences, which is why they are kept distinct. Besides their usage, the most important difference is their sizing behavior. In the case panels, the HL engine tells the panel what its available dimensions are, after which the panel sizes and positions its contents accordingly within those dimensions. For forms, however, the dimensions are dictated by the form itself. Once the dimensions are specified by the form (via its Position script), the HL engine will construct a suitable window that encompasses the form and display it.

Both panels and forms contain one or more layouts and represent a rectangular region comprising the panel/form. The panel/form is responsible for coordinating the position of its contained layouts within its boundaries.

IMPORTANT! Unlike most other visual elements, panels and forms share the same namespace. This means that the unique ids assigned to panels are **not** distinct from the unique ids assigned to forms. You may **not** define a panel with the same unique id as a form, and vice versa. All of the ids assigned to panels and forms must be distinct from one another.

Category: Basic Concepts and Terminology

Sheets

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

Sheets are the top-level visual element used for character sheet output. Sheets define the contents of individual pages of printed output. Sheets contain one or more layouts and represent a rectangular region comprising the sheet. The sheet is responsible for coordinating the position of its contained layouts within its boundaries.

In order to handle lengthy output that spans numerous pages, sheets can be designated as "spillover". The HL engine will automatically track which items have and have not been output, allowing the same sheet to be printed continuously with the "same" contents. With each page, only the material that has not been been printed is included, and output finally stops after all the material has been output a single time. This makes the output of material like spell lists and journal logs extremely easy to manage.

Category: Basic Concepts and Terminology

Scenes

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

The term "scene" is used to generically refer to panels, forms, and sheets. All of these visual elements manage layouts and have numerous similarities in how they behave. While each has its own distinct characteristics, whenever something applies equally to all three, it will be described as applying to scenes.

Category: Basic Concepts and Terminology

Dossiers

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

When HL presents the user with the option to output character information, the list of dossiers is used. This applies to character sheets, statblocks, and even data being exported for use in other products. For character sheets, each dossier is an ordered collection of one or more sheets, which allows individual sheets to be easily re-used across multiple dossiers. For other forms of output like statblocks, dossiers utilize scripts to synthesize the properly formatted text.

Category: Basic Concepts and Terminology

Resources

Context: HL Kit ... Basic Concepts and Terminology ... Visual Building Blocks

All fonts, colors, bitmaps, and borders used within the data files must be defined as resources. These resources can then be referenced by styles for subsequent use by portals.

In order to make it easy to create data files for a new game system, the Kit includes a large assortment of built-in resources that can be immediately put to use. As such, you can largely ignore the need to define visual resources when developing your data files, allowing you to focus solely on getting the underlying functionality into place first. Once the data files are working the way you want, you can then switch your focus back to the resources and being the process of replacing them with something more suitable to the game system.

The Kit also publishes all of the "system" resources that are used by fundamental HL mechanisms and allows them to be modified. This makes it possible to completely replace the system resources used throughout HL, thereby completely transforming the visual look of the entire HL interface. The Mutants & Masterminds data files provide an excellent example of the extent to which you can transform the HL interface.

Category: Basic Concepts and Terminology

Styles

Context: HL Kit ... Basic Concepts and Terminology ... Visual Building Blocks

The Kit makes is easy to manage and tailor the visual look and behavior of the data files through the use of styles. Every portal must specify a variety of colors, fonts, bitmaps, borders, and other facets to be used when displaying that portal. In order to achieve a consistent interface, you'll typically want to use the same basic look and behaviors for most portals of the same type. Similarly, if you want to change the visual look or behaviors of a particular portal type, you'll want to do it for all of them.

To keep things convenient and easy, you'll define styles that encapsulate the common visual behaviors for each portal type. Each portal is then assigned a style that dictates its visual behavior. Changing the behavior for a portal requires simply changing the assigned style, while changing the behavior of all portals of a type requires simply changing the style definition itself.

You can also dynamically change the style assigned to a portal at run-time via scripts. This allows you to customize the visual look of a portal based on conditions determined at run-time, such as flagging something in red that is determined to be invalid.

Category: Basic Concepts and Terminology

Sort Sets

Context: HL Kit … Basic Concepts and Terminology … Visual Building Blocks

The Kit utilizes a "sort set" to appropriately sort the contents of a list of things or picks. Each sort set is essentially an independent sort specification. You can define any number of sort sets and they are not officially associated with any particular lists of objects. As such, you can easily re-use a particular sort set in a variety of situations.

The purpose of a sort set is to spell out the exact rules to be used for sorting a collection of picks or things. Each sort set will consist of one or more sort criteria, and those criteria must be specified in the exact sequence that you want HL to sort the items in.

Each sort criteria consists of a few characteristics. The first aspect is the unique id of either a tag group or a field. The second aspect is whether to sort the items in an increasing order or a decreasing order.

If a tag group is specified, then the sort criterion instructs HL to sequence all items based on the sequencing rules set forth for the tag group. All items that possess a tag from the tag group are sorted based on the tag group's sequencing rules, and any items that lack any tag from the sort group are sorted last.

If a field is specified, then the sort criterion tells HL to order the items based on the results of comparing the values of the designated field. The comparison rules use a simple numeric comparison when the field is a value and a string comparison when the field is text-based. If a particular item does not possess the field for some reason, then that item is always placed at the end of the sorted list.

When multiple sort criteria are specified, they are processed in the order given until one yields a difference, at which point all further sort criteria are ignored.

The starting data files provided by the Kit include a number of pre-defined sort sets. These sort sets provide an excellent starting point and are already used within the starting data files. You can readily revise and extend the set provided for your own needs.

Category: Basic Concepts and Terminology

Encoded Text

Context: HL Kit ... Basic Concepts and Terminology ... Visual Building Blocks

While not exactly a true "building block", encoded text is an important facet of the visual presentation for any game system. Encoded text is the term used for inserting control over colors, fonts, and even bitmaps within text strings. If you want to highlight a word within a string in red, you'll use encoded text. If you want to change the font size for a few works or put a keyword in bold, you'll use encoded text. You can also insert bitmaps into the text stream with encoded text, which makes it possible to do things like display the sequence of dots for traits within the World of Darkness game system.

The encoding syntax uses the characters '{' and '}' to identify the special codes, much like HTML uses the '<' and '>' characters to wrap special codes. The text found between the '{' and '}' characters is interpreted based on the table given below.

{br}	Inserts a carriage return into the text at this position.
{nbsp}	Inserts a non-breaking space into the text at this position. Word-wrapping behavior is not allowed to occur on a non-breaking space, so the space is tied to whatever other text appears before and/or after it. This can be useful for placing padding around text that changes foreground and/or background colors.
{spc}	From this point forward in the text, all sequences of multiple spaces are collapsed into a single space.
{b} {/b}	Text following the "{b}" sequence is rendered in bold. This persists until the "{/b}" sequence turns off bold rendering.
{i} {/i}	Text following the "{i}" sequence is rendered in italics. This persists until the "{/i}" sequence turns off italics rendering.
{u} {/u}	Text following the "{u}" sequence is underlined. This persists until the "{/u}" sequence turns off underline rendering.
{font name}	From this point forward in the text, the font with the specified "name" will be used for output. The prevailing size and style are used in conjunction with the new font. **NOTE!** Use of a font that is not a standard Windows font will result in the new font being ignored on any computer that does not possess the font. So be sure to only use standard Windows fonts if you plan on distributing your data files to others.
{size value}	From this point forward in the text, the font size is changed to the given "value". In order to support fractional point sizes, the value is the number of quarter-points, so a value of 40 would translate to a point size of 10, while a value of 38 would translate to a point size of 9.5. The prevailing font and style are used in conjunction with the new size.
{revert}	From this point forward in the text, the original default font characteristics are restored. All states for bold, italics, and underline are reset to off, regardless of the default font characteristics restored. The primary use for this encoding is to output some text in the default font, configure the text properly for a short segment, and then revert to outputting the rest of the text in the original font. This way, the same encoded text works consistently even when the default font varies between situations.
{text xxxxxx}	This sequence changes the foreground text color to the color given by "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the encoding "{text ff0080}" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80". To revert the foreground text color back to the default, use a color value of "010101". **NOTE!** For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.
{back xxxxxx}	This sequence changes the background text color to the color given by "xxxxxx". The color format uses standard HTML syntax, as described for "{text}" above. To disable use of a background color and display text transparently, use a color value of "010101".
{align position}	From this point forward in the text, each new line of text will be aligned based on the given "position". The position must be one of the following values: "left" for left-aligned text, "right" for right-aligned text, or "center" for centered text.
{vert value}	Vertical spacing is immediately inserted into the output, with "value" indicating the number of pixels worth of gap to insert.
{horz value}	Horizontal spacing is immediately inserted into the output, with "value" indicating the number of pixels worth of gap to insert.
{offset value}	From this point forward, every new line of text has "value" pixels of horizontal spacing inserted at the start of the line. The offset persists, allowing large blocks of text to be inset from other text. You can turn off the offset by applying an equivalent negative offset. Repeated uses of "offset" are cumulative.

{indent value}	From this point forward, every new paragraph of text gains an indentation behavior dictated by "value". If "value" is positive, normal indentation logic is used and the first line of each paragraph is indented that number of pixels. If "value" is negative, a hanging indent is applied, with the second and subsequent lines being indented the specified number of pixels (as an absolute value). Repeated uses of "indent" are cumulative, except for a value of zero, which turns off any active indent behavior.
{bmp name}	Inserts into the output the bitmap image with the file name of "name.bmp" that resides in the data file directory for the game system. This bitmap is assumed to be transparent, with the pixel at (0,0) indicating the transparent color. For example, the code "{bmp foo}" would insert the bitmap file named "foo.bmp" that is found in the game system directory. **NOTE!** If the bitmap is not found or the output doesn't support encoding, the "name" is inserted as text. So if the filename is "[R].bmp" and referenced as "{bmp [R]}", the text inserted would be "[R]". **NOTE!** By default, bitmaps are never scaled when output, but individual styles can explicitly enable scaling of bitmaps inserted via encoded text. If scaling is enabled, the bitmaps may not look ideal due to the scaling.
{bmpscale factor name}	As {bmp name}, above, except the the bitmap is scaled by a factor of "factor" (from 2-9) before being output. For example, if you create a bitmap "somebitmap.bmp" with a size of 400x400 and then output it using {bmpscale 4 somebitmap}, it will be drawn with a size of 100x100. This can be useful for drawing bitmaps at large scales, then shrinking them down for use on high-dpi mediums (such as printed pages). **NOTE!** This is only recommended for use on output sheets. Due to the windows bitmap resizing algorithm, results of this will likely be poor if used on the screen.
{meta behavior}	The behavior specifies a new rendering behavior that will persist until it is again changed. The specified behavior must be one of the following values: ▪ bmpfull – Bitmaps embedded in the encoded text are vertically centered within the full height of the text, including the region of any descender. ▪ bmpbaseline – Bitmaps embedded in the encoded text are vertically centered between the baseline of the text and the top edge of the text. This is the default behavior used by encoded text. ▪ revert – All behavior changes applied via "meta" special codes are reverted to their default state.
{macro name}	Looks for the macro with the unique id given by name and processes the contents of that macro as if it were found in the text instead. This means that macros MAY include encoded text that will be properly processed when the macro is evaluated. **NOTE!** Macros MAY be nested within each other, but only to a limited extent, so use nesting sparingly. If a macro references another macro within it, the nested macro will be correctly processed, up to a small number of nesting levels. NOTE! If an undefined macro is referenced, a message of the form "[Undefined macro: name]" is inserted into the resulting text where the macro would be.
{ref name}	Designates the start of a reference region (or "hot zone") that the user can click within to obtain additional information. The name specifies the predefined reference to show when the user clicks in the zone. The zone spans all text from this point forward until the zone is terminated. Terminating a reference zone is accomplished by leaving the name blank. The following is an example of using a reference: "before {ref foo}hot zone{ref} after". **NOTE!** If an undefined reference is used, a message of the form "[Undefined reference: name]" is displayed when the reference hot zone is clicked within by the user.
{url name}	Designates the start of a hot zone corresponding to an internet URL. The name specifies the actual URL to which the user will be sent when the zone is clicked within (e.g. "http://www.wolflair.com"). The zone spans all text from this point forward until the zone is terminated. Terminating a URL zone is accomplished by leaving the name blank. The following is an example of using a URL zone: "before {url http://www.wolflair.com}hot zone{url} after". **NOTE!** When clicked, Windows is instructed to launch the default web browser that is currently configured by the user and to load the specified URL into that browser. If no browser is properly configured, an error will be reported. If the URL is invalid, the web browser will report that to the user.
\n	Identical to "{br}". This is a programming convenience for automated conversion programs written in C/C++.

IMPORTANT! Not everything within HL makes use of encoded text, so be sure to check the documentation first. If you use encoded text in a place where it's not supported, you will see your embedded special codes within the text displayed.

NOTE! Encoded text sequences ignore case, so "{BR}" is identical to "{br}". Exceptions to this are macro and reference names, which are case-sensitive. Similarly, URLs are case-sensitive if the website being addressed uses case-sensitive URLs.

NOTE! If the text within an encoding does not correctly match one of the codes defined above, the text is left unchanged, except as noted above.

Category: Basic Concepts and Terminology

The Physical Files

Context: HL Kit ... Basic Concepts and Terminology

Overview

There are an assortment of different file types involved in a complete set of data files. These files must reside in specific locations for HL to properly find and use them. In addition, there are critical naming conventions that the data files must subscribe to.

Topics

The various details that govern the physical data files are outlined in the topics below.

- Data File Types
- File Locations and Naming Conventions
- File Loading Order

Category: Basic Concepts and Terminology

Data File Types

Context: HL Kit ... Basic Concepts and Terminology ... The Physical Files

HL utilizes a variety of different file types for defining all the particulars of a given game system. Each of the various file types is summarized below, with their specific contents being specified in the Kit Reference section.

definition file	The definition file provides the basic foundation for a particular game system, defining the fundamental characteristics that are shared across all aspects of that game. Each game system has exactly one definition file. Definition files are only utilized when creating data files for a new game system from scratch. See the Kit Reference section for details.
structural file	Structural files enable you to define the underlying structure of the game system, creating the framework on which all of the visual pieces and game mechanics elements will be defined. You can have any number of structural files, allowing you carve up the logic across multiple smaller and more manageable files. See the Kit Reference section for details.
data file	Data files are where all of the user-manipulated elements are specified for a game system. This includes both the visual elements (e.g. panels and forms) and the game system elements (e.g. classes, spells, skills, and equipment). Data files build upon the material in the structural files. See the Kit Reference section for details.
configuration file	When package files are used, the configuration file contains basic details about the game system, allowing HL to display appropriate entries for the game system for user selection. If packages are not used, then the configuration details always reside within the definition file and no configuration file should be present.
package file	Package files are pre-built collections of data files that are distributed by Lone Wolf Development. Packages provide a convenient way to distribute material to users, as well as offering security-restricted access. A package file that your license is not authorized to access will simply be ignored by HL when loading files. If your license grants access to a package, the package is equivalent to having all of the data files present within the game system data directory.

IMPORTANT! Although there is a specific file type of "data file", the general term "data files" is often used to collectively reference ALL of the various files which are created for a particular game system. Therefore, a reference to the "d20 System data files" would be referring to any definition file, structural files, data files, package files, and/or configuration file for the d20 System game system. When a reference is exclusively referring to a data file for a game system, it will either be spelled out within the Kit documentation or be clear from the context of the reference.

Category: Basic Concepts and Terminology

File Locations and Naming Conventions

Context: HL Kit ... Basic Concepts and Terminology ... The Physical Files

Beneath the directory where you installed HL is a directory named "data". If you installed HL to the default location, this directory will be located at "C:\HeroLab\data". All HL data files reside in sub-directories beneath the "data" directory, with all of the data files for each game system being grouped together within a sub-directory named appropriately for that game system. For example, all of the data files for the d20 System will be found in the "d20" sub-directory, or "C:\HeroLab\data\d20" if you installed HL to the default location.

Within a game system directory, there will be an assortment of files with a variety of names. Here is a summary of the various files that will be found and their purpose.

config.cfg	Configuration file that provides details when package files are used
*.pkg	Package files
definition.def	Definition file that is compiled before anything else
*.1st	Structural files that are compiled first
*.core	Structural files compiled after "*.1st" files
*.str	Structural files compiled after "*.pri" files
*.aug	Structural files compiled after "*.pri" files
*.dat	Data files that are compiled after structural files
logo.bmp	Bitmap file containing game system logo
*.bmp	Image files used for buttons, textures, etc.
*.hlz	Compiled files used by HL at run-time
faq.htm	HTML file containing the FAQ for the data files that is generated when the data files are compiled
manual.htm	Manual with specifics on using the data files for the game system (file name may be different)
timing.xml	Timing dependency and error report generated whenever the data files are compiled
unused.xml	Report of unused resources and styles that is generated when the data files are compiled

Category: Basic Concepts and Terminology

File Loading Order

Context: HL Kit ... Basic Concepts and Terminology ... The Physical Files

As you may have spotted within the previous section, the Kit supports four different types of "structural" file. The information that can be placed in these files is identical, so this begs the question of why there are four types. The reason is that HL loads files in a specific sequence and all the material in one set of files must build on the previous set.

You can think of it as layers, where each new layer relies upon the layers beneath to be correct. During loading, each new layer of data files requires that certain information it references already be in place and verified. For example, tag groups that are used must be defined prior their use, else an error is reported.

In order to keep your data files more manageable, you'll likely want to keep them small and focused. Having a few huge files is perfectly acceptable, but we recommend against that approach and have structured the example data files accordingly. However, maintaining a variety of smaller structural files means that you'll be carving the information up across different files, and that can result in references across files that aren't defined.

Using tag groups as an example again, you might define a tag group in one file and then reference it in another. The problem is that you need to make sure that the file in which the tag is defined is always loaded **before** the file in which it is referenced. The only way to do this reliably is for HL to support different structural file extensions and load those files in a specific sequence. This way, you can put the tag groups in one file that you know will be loaded first and the references in other files that you know will be loaded afterward.

You can have lots of different structural pieces in a complex set of data files. And there are a variety of requirements regarding what information must be defined before it is used. Consequently, in order to handle complex situations, there are a four different structural files. Putting all together with the other data file types, there are a total of six layers of files and they are loaded in the sequence specified below.

1. Definition file (definition.def)
2. Structure files with the ".1st" file extension
3. Structure files with the ".core" file extension
4. Structure files with the ".str" file extension
5. Structure files with the ".aug" file extension
6. Data files with the ".dat" file extension

Category: Basic Concepts and Terminology

Data Manipulation Basics

Context: HL Kit … Basic Concepts and Terminology

Overview

In order to handle the complexities of every game system, the HL engine is a very sophisticated tool. However, we've invested significant time and effort to keep things as simple and streamlined as possible. The net result is that the Kit makes extensive use of a few powerful mechanisms that give you complete control over both how data is manipulated and when it is manipulated.

Given the volume of information involved in many game systems, data manipulation entails the proper sequencing of tasks. The first key mechanism is the evaluation cycle that governs when the data is manipulated. The second key mechanism is a fast and flexible classification system, called tag expressions, which uses tags to identify which objects are to be manipulated and which manipulations should be applied to them. The final key mechanism is the scripting language that allows the actual manipulation of the data.

Topics

Each of the above mechanisms is introduced in the sections below. An in-depth discussion of these mechanisms is provided in separate sections of the documentation.

- Evaluation Cycle Basics
- How Tags Work
- Tag Expression Basics
- Scripting Basics
- Information Access
- Types of Scripts

Category: Basic Concepts and Terminology

Evaluation Cycle Basics

In order to ensure that all data manipulation operations are applied in a correct and consistent order, the HL engine enforces a strict sequence of evaluation that you completely control as data file author. This sequence is triggered repeatedly as the user makes changes to characters, and it is referred to as the "evaluation cycle".

Whenever the user takes any action, HL automatically re-evaluates all facets of the portfolio that are impacted by the change. This updates all dependencies on the user's changes. For example, if the user modifies an ability score in the d20 System, all linked skills and dependent weapons are updated to reflect the impact of the change. To safeguard against lag time in the user-interface when the user takes multiple actions in rapid succession (e.g. clicking the '+' button to increment a skill rating a dozen times), HL waits until the user pauses for a moment before it initiates a new evaluation cycle on the portfolio.

Everything that is performed by the Hero Lab engine is done in a specific sequence, and the majority of actions occur during the evaluation cycle. Each of these individual actions is referred to as a "task", and the complete set of tasks comprising the evaluation cycle is referred to as the "task list". For virtually every task, the data file author controls the evaluation sequence by designating when each task should be processed. There are two criteria used to determine the scheduling of a task: phase and priority.

For each game system, the data file author defines a set of phases that dictate the general sequence in which evaluation is performed. Each phase typically corresponds to a logical step in the overall evaluation cycle, such as "initialization", "before level-based calculations", "after attribute modifiers", etc. All phases are ordered, thereby dictating the sequence in which the phases are processed during the evaluation cycle.

Every task is assigned a phase during which it will be evaluated. All tasks are also assigned a priority, which controls the order in which tasks within the same phase are processed. If two or more tasks have the exact same phase and priority, then the engine uses a number of rules to order them. If the two tasks are still scheduled for the same time, the engine is free to schedule them in whatever sequence it finds convenient, and this order **may change** from one evaluation pass to the next. Consequently, assigning the correct phase and priority is often critical to ensure that modifications are applied before or after subsequent tests are made that rely on those modifications.

The Kit provides an assortment of pre-defined phases that should serve as an excellent starting point for just about any game system. You are free to change or delete these phases, as well as add new phases to suit your needs. However, the set provided will typically work well for most game systems we've encountered.

NOTE! When the evaluation cycle begins, it continues until completion. This is usually transparent to the user, but it can become noticeable on older (i.e. slow) computers when the data files are highly complex. Therefore, it's best to utilize tag expressions whenever possible to limit the number of objects that must be processed during evaluation. Similarly, its typically faster to use tags instead of scripts when possible, because they are significantly faster to execute.

How Tags Work

Context: HL Kit … Basic Concepts and Terminology … Data Manipulation Basics

Virtually everything within the Kit leverages tags in some way. Tags represent a generic mechanism for assigning arbitrary attributes to a thing, pick, or container. The only restrictions regarding tag assignment are those that are defined for a given tag group (e.g. minimums, maximums, duplicates, etc.). This allows the data file author to design the set of tags and tag groups specifically for the behaviors of a particular game system. With this simplicity comes a great deal of power and flexibility. Unfortunately, the lack of a regimented structure also introduces some potential confusion when getting started. The rest of this section will hopefully point out some useful pointers to get you underway.

We'll start with an in-depth review of the tags model. To use a real-world analogy, consider a database of shirts. Every shirt has a color, a sleeve length, a size, a style (e.g. t-shirt, polo shirt), and various other characteristics. Each of these would translate to a tag that is associated with a given shirt. There would be one group of tags for the color, with a separate tag for each color. Another group would reflect the size, with separate tags for each size. So all you need to do is associate the proper tags with the shirt to describe it. Collectively, all the tags assigned to the shirt comprise its description.

So what if you are presented with a shirt that is striped in red and white? Do you classify the shirt as red or white? Your first thought might be to define a new group of tags for the design of the shirt, with possible tags including stripes, polka dots, plaids, etc. But that still doesn’t address the issue of color. The goal is to easily search for shirts in your database, so the best choice is to declare the shirt as both red and white, assigning it both tags. This way, if the user is looking for a shirt, the user can decide whether he wants a shirt that is only red, a combination of red and another color, or specifically red and white. Using tags, you can easily make this distinction.

Let’s take this a step further. What if you have some shirts with two-colors of stripes and other shirts with three (or even four) different stripes? One option would be to have different design tags for each number of stripes. But what if a user just wanted to find all the red striped shirts, without worrying about how many stripes? The tags method provides two ways of easily handling this situation. The first technique is to assign the same stripes tag to the shirt multiple times to indicate the number of stripes on the shirt. This allows the user to query striped shirts in general (at least one stripes tag assigned to the shirt) and shirts with varying numbers of stripes (the shirt must have X number of stripes tags assigned).

The second technique is to define a separate tag for each number of stripes, but do it so that wildcards can be utilized. For example, defining the tags “stripes2”, “stripes3”, and “stripes4” would make it possible to assign the proper tag corresponding to the number of stripes on the shirt. The user could query striped shirts in general by using a wildcard to identify any tag that starts with “stripes”. Querying for an exact stripe count would simply look for shirts with the specific stripes tag.

All of the scenarios described above will occur regularly when designing data files for role-playing games. They will be useful for validation rules, condition tests, and managing when and how to apply adjustments to picks and containers. But the key detail is that the tags mechanism provides a single, generic mechanism that can be creatively used to handle virtually all design situations – both efficiently and without a high level of complexity for the author. The trick is in picking the right set of tags to make your life easier as a data file author.

To help you with that last consideration, the starting data files included within the Kit provide an assortment of tags and tag groups to handle many of the most common situations. These tags are all utilized within the starting data files, so you can see concrete examples of how they can be put to use.

Category: Basic Concepts and Terminology

Tag Expression Basics

Context: HL Kit ... Basic Concepts and Terminology ... Data Manipulation Basics

Tags form a fundamental building block upon which much of HL is constructed, and tag expressions are where they become of critical importance. Since the vast majority of objects you'll be managing are things and picks, there must be a way to identify the proper subset of these objects that apply to a particular situation. For example, attributes, skills, and weapons are used in completely different ways, so you want to keep them separate from each other – yet they are all things (or picks). The solution is to assign tags to each object and then use a tag expression (or tagexpr for short) to identify the subset of objects that apply in a given situation. A major (separate) section of the documentation is dedicated to the subject of tag expressions, but a brief overview is valuable at this point.

A tag expression is essentially a filter that gets applied to all objects of particular type (e.g. things or picks) and selects only the ones that meet the specified criteria. Tag expressions are Boolean expressions, which means they evaluate to a simple "true" or "false" result. They examine all of the assigned tags and determine whether those tags satisfy the expression or not. Separate criteria can be combined, allowing you to require that multiple criteria all be met, one of a set of criteria be met, certain criteria be excluded, or some combination thereof. For example, a tag expression could test whether a thing has the tag "Elven" from the "Language" group. Or a more complex tag expression could test whether a thing has the "Language.Elven" tag and also has either the "Race.Elf" or "Race.HalfElf" tag.

Since tag expressions can utilize full Boolean logic (i.e. "and", "or", "xor", "not") and can even extract and test numeric values from tags, tag expressions can model extremely complex conditions without difficulty. The bottom line with tag expressions is that they provide a powerful and flexible method for quickly determining whether to include or exclude an object, and they are based exclusively on the set of tags assigned to that object. As such, they are used extensively throughout HL.

Category: Basic Concepts and Terminology

Scripting Basics

Context: HL Kit … Basic Concepts and Terminology … Data Manipulation Basics

Contents

Overview

HL makes extensive use of scripts to allow the data file author substantial freedom and flexibility. In fact, scripting is such a fundamental and diverse topic that huge sections of the documentation are dedicated to various facets of writing scripts. This section merely provides a brief overview.

The scripting language syntax within the Kit is relatively simple. You can declare variables, assign values, perform simple conditional tests, and utilize a number of built-in intrinsic functions for various purposes. There is also a syntax that allows access to all of the objects within a given actor, such as the various picks and containers, plus the field values and tags that may be assigned to them. The language syntax itself is somewhat similar to the age-old Basic language. Using scripts, an author can pretty much do whatever is necessary to properly model the requirements of a given game system.

To make the writing of scripts easier, the Kit supports re-usable procedures. A procedure is nothing more than a mini-script that can be called from multiple places. The data files for each game system make extensive use of procedures so that many scripts can be reduced to simply calling one or two procedures to do all the work.

The starting data files provided by the Kit include numerous scripts and procedures that you can use, adapt, and extend according to the needs of the game system you're implementing.

Thing-Based Scripts

The most common type of script that you will find yourself writing the "eval script". Of all the different types of scripts, most are triggered in response to a specific action. However, an eval script is scheduled to occur at a specific phase and priority during the evaluation cycle, hence the name.

Every eval script is associated with a thing. This means that every pick derived from a given thing inherits all of the eval scripts for that thing.

Since an eval script is performed during evaluation processing, a separate task is always created for each eval script, which is then scheduled within the overall task list. If a thing is added to a container multiple times, each eval script must be processed separately for every pick. Consequently, separate eval script tasks are created and scheduled for every pick.

Eval scripts are the most commonly used script due to the ability to schedule them. So many facets of a complex game system are inter-dependent. Dependent calculations must be performed in a carefully ordered sequence to ensure that all of the game mechanics are accurately implemented within the data files. Eval scripts provide the means for this scheduling.

Thing-Based Rules

Sometimes, you don't need to apply a modification to anything and instead need to verify whether a required condition is satisfied. For example, the "Knowledge" skill might require that the user specify the domain to which the skill applies. This entails checking that the user has entered a suitable value.

To handle situations like this, the Kit provides "eval rules" as a companion to eval scripts, and you will likely find yourself writing a number of these as well. Just like eval scripts, eval rules are scheduled as tasks and performed with specific timing during the evaluation process. Unlike eval scripts, the purpose of an eval rule is to verify a condition, so an eval rule must always determine whether its condition is satisfied or not. If the rule is not satisfied, the corresponding pick is flagged as being invalid and an appropriate message is displayed to the user within the validation report.

Component-Based Scripts and Rules

There are many situations where you'll want the same script or rule to be applied to all things that derive from a particular component. For example, every piece of gear the character is wielding should be verified to not be stored in a backpack or some other container. The Kit makes it easy to handle situations like this by allowing you to actually define the script or rule directly on the component.

Whenever a script or rule is defined on a component, that script/rule is treated as if it were individually assigned to every thing that derives from the component. This means that a new task is automatically created and scheduled for the script/rule on every pick that is added to the character.

It is reasonably common to have both component scripts/rules **and** individual script/rules associated with a particular thing.

Category: Basic Concepts and Terminology

Information Access

Context: HL Kit … Basic Concepts and Terminology … Data Manipulation Basics

Contents

Overview

The goal of writing a script is to retrieve and/or manipulate one or more facets of either an actor or a visual element. Consequently, your primary focus, as an author, will be on how to access information. The Kit uses an hierarchical structure for managing all the different objects. This section of the documentation outlines the basics of how to navigate the hierarchy and access the information you need.

Script Contexts and Transitions

Every script begins with an initial *context*. The context depends entirely upon the script, but the context always refers to a specific object maintained by HL, such as a pick, an actor, or a layout. For example, an Eval Script begins with an initial context of the pick that the script will be executed upon. Similarly, a Position script for a layout begins with an initial context of the layout for which the contents need to be positioned on the screen.

When you are writing a script, you will often need to retrieve information from or apply changes to objects other than the initial context your script begins with. This is accomplished by performing a *context transition* to the context of a connected object within the hierarchy, thereby establishing the new object as the new context. For example, you can transition from a pick context to the context of the container that contains the pick. Or you can transition from a pick context to the context of one of its fields.

Using context transitions, you can "travel" through the hierarchy of objects (within limits), seeking out the specific objects you need to manipulate via your scripts. The exact list of what context transitions are possible is entirely dependent on the script and the current context. A complete list of contexts and specific transitions available is outlined in the Kit Reference documentation.

IMPORTANT! The initial script context resets for each separate identifier used within a script - it does not persist. This makes it easy to have a single statement access multiple different contexts by transitioning in different directions from the same initial starting point.

NOTE! It is possible to transition to an invalid (i.e. non-existent) context. For example, a script might try to transition from a container to a pick that doesn't exist within that container. If that occurs and the non-existent context is then accessed (e.g. getting the name of the non-existent pick), a run-time error will be reported and the access will fail.

Script Targets

Once you have established the final context to operate upon, you need to identify the specific facet of the context to access or manipulate. This is referred to as a *target reference*, or sometimes simply *target*. For example, if you have a pick context, you must specify whether you are interested in its name, its tags, or something else.

The syntax for accessing a given target aspect will vary, depending on what the target is and what information is needed to uniquely identify the desired target. All of the different target references are defined in Kit Reference documentation.

Target Identifiers

In order to access or manipulate a component, your script must specify both the intended context and the target. Scripts use an open-ended dot notation to convey this information and uniquely designate a *target identifier*. A target identifier represents the combination of context and target reference, which should yield a specific piece of information to operate upon. You can use valid target identifiers anywhere within scripts that you would use a simple variable, allowing you to retrieve the current value of a target identifier or set that value.

The target identifier consists of a sequence of zero or more context references, each separated by a period ('.'), and finally followed by a singe target reference. In sequence, each context reference must indicate a valid context transition from the previous context. The target reference must be valid for the final context established. This syntax for specifying a target identifier is referred to as the *dot notation*.

Category: Basic Concepts and Terminology

Types of Scripts

Context: HL Kit … Basic Concepts and Terminology … Data Manipulation Basics

The Kit supports more than 40 different types of scripts that make it possible for you to control virtually every facet of how the data files behave. Some scripts control visual facets of how information is presented, while others control the actual manipulation of the character. In general, all of these types of scripts can be distilled into a handful of different categories.

The list below presents a summary of the various categories of scripts, while the complete list of all script types will be found in the Kit Reference documentation.

Visual Positioning	Scripts that manage the size and positioning of visual elements within panels and sheets
Synthesis & Presentation	Scripts that synthesize information for display to the user in some fashion, including labels, descriptions, mouse-over information, and stat blocks
Pick Manipulation	Scripts that manipulate the contents of picks during the evaluation cycle
Field Manipulation	Scripts that manipulate the contents of fields for both display and constraint
Validation	Scripts that apply validation tests to objects with integrated reporting of errors
Creation/Deletion	Scripts that perform appropriate setup and cleanup of specialized objects
Transaction	Scripts associated with the buying and selling of equipment
Mode Transition	Scripts associated with the transition into and out of advancement mode
Trigger	Scripts that are invoked in direct response to user actions, such as merging and splitting stackable gear, controlling combat and turns, etc.
Fixup	Scripts used to accommodate changes between data file releases and potential loading errors of portfolios

Category: Basic Concepts and Terminology

Manipulation of Visual Elements

Context: HL Kit … Basic Concepts and Terminology

Overview

In addition to getting the underlying game mechanics correct, you'll also need to effectively present the information to the user and allow him to create a character. The overall hierarchy of visual elements has already been described. This section delves into the process of successfully manipulating those visual elements to achieve an interface that works smoothly.

Topics

There are various separate topics associated with the manipulation of visual elements. Each of these topics is discussed in the sections below.

- Screen Vs. Print Output
- How Visual Elements Behave
- The Positioning Sequence
- Positioning Portals Within Templates
- Dynamically Changing Styles
- Keyboard Tab Order
- Working With Tables
- Panel Display Order
- Using Automatic Placement

Category: Basic Concepts and Terminology

Screen Vs. Print Output

The Kit makes a critical distinction between output to the screen and output to the printer. Similarly, there are completely separate portals for the purpose of screen output versus for printed output. Layouts and templates are classified for screen versus print output based on their contents. Panels, forms, and sheets are only allowed to contain layouts that are designed for screen or print output, as appropriate to their nature. Sheets are intended for print output, while the other two are intended for screen output. This begs the question of why the distinction is made.

The obvious distinction is that portals used on the screen will typically involve some means for the user to modify content, while portals for printed output are for display only. However, there are a number of on-screen portals that are solely for display purposes (i.e. labels, images, and fixed tables). In fact, those on-screen portals are the exact ones that have equivalent versions for printed output, so why couldn't those portals be the only ones allowed for output use? Because there is more going on that may not be immediately obvious.

The key reason for the separation is that portals for printed output have some critically different needs from on-screen portals. By keeping the portals distinct from each other, those different needs and behaviors can't be inadvertently confused by authors. And by percolating that distinction up through the entire visual hierarchy, everything is kept clear for authors.

So what are some of these important distinctions? A quick sampling is provided below.

- Portals for output have different style needs from on-screen portals. Whether it be the use of color, special borders, or something, the visual for printed output are significantly different from on-screen display.
- Different portals needs to behave differently within printed output. For example, the titles drawn above major sections of printouts don't work the same and have different needs from the titles drawn above sections on-screen.
- For printed output, it is commonplace to need items of varying height within tables. On the screen, it's more valuable to use fixed height items with mouse-over regions so that everything is kept compact and easily scrolled through for long lists.
- The scaling logic used for images can be different when displaying on a comparatively low-resolution screen versus on a high-resolution printer.

The list goes on, but this should give you a good idea of why the distinction between printed and on-screen output is important.

How Visual Elements Behave

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements

Contents

Overview

Visual elements share a number of behaviors that are common across most, if not all, different visual types. These common behaviors are discussed in the topics below.

Rectangular Region

Every visual element represents a rectangular region, whether it be on the screen or on the printed page. As such, each element possesses both dimensions and a position. The dimensions are a simple width and height. The position is given as the location of the top left corner of the visual element within its containing visual element.

The position is always relative to the containing element and consists of two values. It represents a number of pixels along the X axis and a number of pixels along the Y axis. Values along the X axis increase as you move from the left to the right. Values along the Y axis increase as you move from the top to the bottom. By convention, a position always lists the adjustment along the X axis first, so a position of (7,42) indicates 7 pixels to the right along the X axis and 42 pixels down along the Y axis.

As an example, consider a portal that is placed at position of (10,0) within a template. Then the template is placed at a position of (5,0) within a layout. The layout is placed at a position of (2,0) within a panel. Putting it all together yields an effective position of (17,0) for the portal within the panel.

When modifying the rectangular region for a visual element, there are two different ways to accomplish it, and each yields different results. The first method is to set the width and/or height for the element. This keeps the top left position anchored in place and merely moves the bottom right position. The alternative is to directly modify the right and/or bottom edges of the rectangle, which implicitly changes the dimensions assigned to the visual element.

Margins

All visual elements that can contain other elements possess margins. The margin is a gap that is maintained between the outer edges of the visual element and the interior region where child elements are positioned. A vertical margin is maintained separately from any horizontal margin, and both default to zero.

For the purposes of placing child elements within a visual container, the zero position is equal to any margin value. Normally, when placing a portal at a position of 5 within a template, the portal appears at the position of 5. However, if the template has a margin of 3, that gets added to the specified position. This yields an effective position of 8 for the portal within the template.

Margins accumulate at every level within the hierarchy. Let's return to the example used in the previous topic about a portal within a template within a layout within a panel. If each of the visual contains has a margin of 3, then each successive child element is shifted 3 pixels further to adjust for the margin. This yields an effective position of 26 for the portal within the panel.

Visibility

Every visual element has a visibility state, which can either be visible or hidden. By default, all visual elements start out as visible. However, you can set the visibility dynamically within any script that positions a portal. If a visual element is designated as hidden (i.e. not visible), then all visual elements within that element are hidden as well. This continues down through the hierarchy. Consequently, setting a layout to non-visible implicitly hides all templates and portals within the layout, plus it also hides all portals within the now-hidden templates.

There will be many places where you will want to control the visibility of visual elements.

Category: Basic Concepts and Terminology

The Positioning Sequence

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements

Contents

The Position Script

Every visual container possesses a "position" script, which means the script is possessed by every template, layout, panel, and sheet. The Position script serves one specific purpose - to properly size, position, and configure all of the child elements within the visual container. Not every visual container needs to define a position script, since the HL engine performs some actions automatically, but most visual containers will perform at least some operations within their Position, and some will perform extensive operations to properly setup the visual elements within them.

Recursive Descent Through Hierarchy

The HL engine utilizes a consistent process for positioning all of the visual elements. Each top-level element (i.e. scene) is handled independently. Within the context of each scene, a recursive descent is performed upon all of the visual elements it contains, invoking the Position script within each visual element during the descent. How this works is detailed in the sections below.

Prior to doing the recursive positioning, all visual elements within the scene are properly reset. This entails setting their position to (0,0), performing default sizing, and initializing their default state. In general, only portals possess default sizing and state.

Positioning Scenes (Panels, Forms, and Sheets)

All positioning starts with a top-level visual element: either a panel, a form, or a sheet. Within the scene, the Position script is invoked. After the script returns, any child layout that has not yet been rendered is now rendered.

Within the Position script, the panel or sheet may need to force a layout to properly calculate its size before another layout can be positioned relative to the first one. When this occurs, the panel or sheet can explicitly tell the layout to render itself by using the "render" target reference on that layout. Rendering a layout invokes the Position script for that layout immediately. Since each layout is automatically rendered at the end, triggering the render from the Position script is only necessary if the rendered results are needed within the script.

Rendering a layout does not place anything on the screen or page. All it does is trigger appropriate sizing, positioning, and state configuration. It is perfectly valid to render a layout multiple times. In fact, there will be situations where you will want to do this to optimally position information. You can render a layout, find out about size of the layout, force a change to the layout, then render the layout again.

NOTE! Rendering a layout multiple times is computationally expensive. Consequently, you should limit re-rendering to only be used when it is truly needed.

Positioning Layouts

When a layout is positioned, its Position script is invoked. Once the script returns, any child templates that have not yet been rendered are now rendered.

Just like with panels and sheets, the layout may need to force a child template to calculate its size before a separate visual element can be positioned relative to it. You can explicitly trigger a template to be rendered by invoking the "render" target reference on the template. As above, rendering a template immediately invokes the Position script of that template. This is not always necessary, since the layout will automatically render all templates after the Position script completes.

In the same way that layout rendering does not actually output anything, template rendering simply determines the position, size, and state of the template and its contents. Nothing is actually output, so it is valid to render a template multiple times, and there may be situations where you need to do that.

NOTE! Rendering a template multiple times is computationally expensive. Consequently, you should limit re-rendering to only be used when it is truly needed.

Positioning Templates and Portals

When a template is positioned, its Position script is invoked. This orchestrates the sizing and positioning of all portals within the template. Unlike the above visual elements, when the Position script returns for a template, there is nothing more to do for that template and/or its contents.

Category: Basic Concepts and Terminology

Positioning Portals Within Templates

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements

Contents

Overview

Due to the way the Kit is designed, the single most common thing you will find yourself doing with visual elements is positioning portals within templates. To better streamline development, the Kit has a number of special optimizations that can be used to efficiently position portals within templates. The topics below discuss some of these shortcuts.

IMPORTANT! The various target references described below are **only** usable on portals within **templates**. If you attempt to use them on portals within layouts (e.g. tables and choosers), they will not work.

Automatic Sizing

The HL engine includes logic that will intelligently determine the dimensions of portals. For example, a label portal will have its size calculated to match the text its contains, an image portal will be sized to match the image it contains, etc. If an appropriate default size cannot be determined, then something safe is used instead.

This mechanism is referred to as automatic sizing and is accessible at any time via scripts. In addition, you can trigger the default sizing logic to be applied to only a single dimension instead of both width and height. As an example, if you change the style of a portal and thereby change the font size used, you will need to re-size the portal based on the new style. Using the automatic sizing logic is a quick and easy way to handle it.

Default Sizing

At the beginning of the visual positioning logic, every portal is assigned an appropriate default size. This is achieved by determining the automatic sizing characteristics for the portal, as described above. As a result of this default sizing, you often don't have to worry about setting the dimensions of portals and can simply focus on positioning them.

Size of Contents

For text portals especially, but sometimes for other portals, you will want to explicitly retrieve facets of their contents. There are three different types of sizing information that can be retrieved. The first is the "text width", which retrieves the width necessary to contain the full text on a single line. The second is the "font height", which retrieves the height of a single line of text, based on the font characteristics established by the current style assigned to the portal. The third is the "text height", which calculates the number of lines of text needed to display all of the text contents and the full vertical extent of that text.

The "alignedge" Target Reference

When positioning portals, a few will be positioned based on the edge of the template, with the rest being positioned relative to the other portals. When you need to position a portal relative to an edge of the template, you can use the "alignedge" target reference. You identify the portal, the edge (left, top, right, bottom), and the offset to use. The offset makes it easy to leave gaps from the edge if you need them.

The "alignrel" Target Reference

Once you've positioned the initial portals relative to the edges of the template, it's now time to start positioning portals relative to each other. To simplify this process, the "alignrel" target reference is provided. You start with the portal to be positioned and then specify the portal to position relative to. You then add the positioning relationship to be used and any offset to be inserted. The positioning relationship identifies both the edge of the portal being placed and the edge of the reference portal, and any combination is allowed. For example, a "left to right" relationship means that the left edge of the portal being placed is positioned relative to the right edge of the reference portal. If an offset of 10 is given, then the portal would be placed 10 pixels to the right of the reference portal.

The "centerhorz" and "centervert" Target References

You will quite often need to center portals along one axis within the template. To accomplish this, all you need to do is invoke the "centerhorz" or "centervert" target reference on the appropriate portals. The HL engine will calculate the relative dimensions and properly position the portal to be centered along the specified axis.

The "centeron" Target Reference

There are times when you need to center multiple portals along a common central axis. In a situation like this, you must first position one portal where it needs to be. After that, you can position the remaining portals relative to the original portal by using the "centeron" target reference. The HL engine will calculate the center line for both portals and position the new portal appropriately.

The "centerpoint" Target Reference

When setting up columnar tables, you will sometimes want to center data within columns. There are two ways of handling this. First, you can configure each portal to center its contents within itself and then set the width of each portal to match the column width being used. The drawback to this is that the default sizing logic will set each portal to an appropriate width based on its contents, and that means that you'll need to reset the width of every portal and position it properly.

The alternative is to calculate the center position of the column. Once you have that, you can simply use the "centerpoint" target reference to position each portal. The HL engine will determine the width/height of the portal and position it so that it is centered on the point specified.

Category: Basic Concepts and Terminology

Dynamically Changing Styles

Context: HL Kit … Basic Concepts and Terminology … Manipulation of Visual Elements

Every portal is assigned a style, and that style dictates a variety of visual characteristics about the portal. They can include the font size, font style, text color, background color, border, bitmaps used, and an assortment of other attributes. In a nutshell, the style is what gives a portal its visual look and behavior. However, the style is **not** fixed - it can be changed.

There will be many situations where you'll want to highlight a particular condition to the user. If you think of the style as fixed, then you're likely to conclude that the best way to switch from a normal looking label to one that is in red text and a large bold font is to have two different portals. You can then hide whichever portal is not applicable based on the prevailing condition state. While this approach will definitely work, it entails extra effort.

The better approach is to simply change the style used on the portal. Since you'll need to define the alternate style to support a second portal, there is no extra work involved there. Then, within the Position script for the containing template, simply check the condition and, if appropriate, use the "setstyle" target reference on the portal.

The only caveat that you'll need to worry about with this approach is that you have to be aware of changing the font size and/or font style. If you change either of those characteristics with the new style, the default sizing of the portal will become invalid. Fortunately, all you need to do is then use the "autosize" target reference on the portal to trigger the appropriate resizing. Just make sure that you do this **before** you position the portal within the template.

Category: Basic Concepts and Terminology

Keyboard Tab Order

Context: HL Kit … Basic Concepts and Terminology … Manipulation of Visual Elements

Contents

Overview

HL provides full keyboard support for navigating around the interface. This includes the ability to use the <Tab> key to move through the various visual elements via the keyboard.

Tab Order Within Templates

Authors can easily control the order in which portals are moved between within a given template. The tab sequence for portals is simply the order in which the portals are defined within the template. If a portal is listed as the very first portal within the template, then the HL engine assumes that portal should be given the keyboard focus first. If the user presses the <Tab> key, then the next portal defined within the template becomes the new keyboard focus. This process continues until the last portal in the template is reached.

If a portal within the template cannot be given the keyboard focus, the focus goes to the next portal, continuing until a valid portal is found. Some portal types, such as labels, can never receive the keyboard focus. Other portals can be made non-visible or disabled via scripts, which renders them unable to receive the keyboard focus.

Tab Order Within Tables

The tab order within a table gets a bit interesting. The following sequence outlines the logic used for tables.

1. If the table has a header template, then any portals within the header are first assigned the focus.
2. After the header, the table itself is considered as being a selectable portal via the keyboard, so it receives the focus next. This is only performed for tables that are scrollable, since given the focus to the table allows it to be scrolled via use of the arrow keys.
3. After the table itself, control passes to the first item within the table. The template associated with the first item dictates the tab order for portals displaying information about the first item.
4. Control passes through each item in the table, in order, with the template again dictating the tab order for that item.
5. If the table has an add option or footer at the bottom, control passes to it. The template associated with the add option or footer dictates the tab order for portals therein.

Once control passes from the last portal within with the table, it transfers to the layout that contains the table.

Tab Order Within Layouts

Within a layout, the tab order of the various visual elements is controlled by the "taborder" attribute that can be assigned by the author. Both portals and templates are mixed together for the purposes of determining tab order, and they are sequenced in increasing order based on the value assigned.

When a template is next in the tab order, HL will transfer control to the first portal within the template that can accept the focus. The focus will continue through to the last portal within the template. Once the last portal is reached, the focus will be transferred to the next template or portal specified by the layout.

When a portal is next in the tab order, HL will set the focus to that portal. If the portal contains other portals (e.g. a table), then the focus will continue to move through the contents of the portal. Once the final child portal is reached, the focus will be transferred to the next template or portal specified by the layout.

Tab Order Within Panels and Forms

Within a panel or form, the tab order of the contained layouts is dictated by the sequence in which the layouts are specified within the panel. When a panel/form is shown, the first layout is given the focus, and the appropriate rules for layouts are used to establish the initial focus. When the user moves through the last visual element within the layout, the next layout is given the focus, and the process continues. When control returns from the final layout within the panel/form, the focus is given to the first layout, and the entire cycle begins anew.

Category: Basic Concepts and Terminology

Working With Tables

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements

Overview

Table portals behave in significantly different ways from other types of portals. Since you will be using tables extensively in your data files, it's critical that you become familiar with the nuances of working with tables.

Topics

The topics below explain the most important differences that you need to be aware of.

- Controlling Table Contents
- Tables Use Templates
- Sequencing Through Sort Sets
- Adding Items to Tables
- Using Choosers
- Controlling Table Rows and Columns
- Template Sizing Within Tables
- Table Headers and Footers
- Tables Always Auto-Size
- Keying on Items Within Tables

Category: Basic Concepts and Terminology

Controlling Table Contents

The purpose of tables is to display a collection of picks or things that are related in some way. For example, all the attributes for a character are shown in a table, all the skills are shown in a table, all the weapons and armor are broken up across multiple tables, etc. So the first order of business with a table is identifying the set of picks or things to be viewed.

Tables are generally associated with a specific component. This component establishes the first criteria for determining which objects are displayed in the table. If an object derives from the specified component, then it is valid for inclusion in the table. In many situations, this is all that's required.

However, there are times when additional filtering is required. In this case, you can specify an additional tag expression that is applied to every object that derives from the component. This is referred to as the "List" tag expression, since it identifies which objects are shown within the list of objects for the table. Any object that satisfies the List tag expression are included in the table, while any object that does not is omitted.

Tables Use Templates

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements ... Working With Tables

Contents

Overview

Tables make extensive use of templates. All of the objects shown within a table are viewed through a template. If the table is dynamic and allows items to be added, a separate template is used for displaying the items available for selection. If a table has a header or footer, then a template is used to display that information.

In fact, everything displayed as part of a table is accomplished using a template. The reason for this is simple. Using templates makes tables highly modular, allowing authors to readily customize the contents and re-use common templates for similar purposes.

How Templates Are Used

Every item shown within a table is either a thing or a pick. Every item that can be selected for addition to a table is a thing or a pick. Templates are an ideal vehicle for display within tables, since templates are always associated with a specific thing or pick. When used for showing a list of items, the same template is simply associated separately with each item in the list to properly render the contents of each item.

When rendering the table to the display, the HL engine will retrieve the list of objects that belong to the table. Then each of those objects is rendered, one at a time, into the table. For each object, the template defines the portals to be displayed and the object provides the appropriate fields from which to pull the information. Since the template is rendered separately for each item, it's perfectly reasonable to customize the contents of the template for each object. For example, certain portals can be hidden for some objects, while styles can be changes for certain portals on other objects.

The Show Template

Every table uses at least one template - the "show" template. This template is used to display each of the items within the table.

Reusing Table Templates In Multiple Places

Templates are not restricted to where they can be used. Consequently, you could choose to use the same template within a table and outside of that table. Or you could use the same template within multiple tables. Doing this is extremely useful. The problem is that, if you do share the template, you will probably need to make a few adjustments to how everything is handled within the template based on its context. The Kit makes this possible by providing the "intable" target reference that can be used on within the template's Position script. This target reference can be used to determine whether a template is within a table in general or to determine which specific table it is being used within.

Category: Basic Concepts and Terminology

Sequencing Through Sort Sets

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements ... Working With Tables

Tables utilize sort sets to specify the sequence in which objects are displayed. There is no requirement that a sort set be given for a table. If omitted, then all of the items are simply listed in case-independent, alphabetical order.

One sort set is used to dictate the order in which the table contents are shown. However, a separate sort set can also be specified for dynamic tables. Dynamic tables provide the ability to select items to add to the table, and the available items are presented in a list. So the second sort set controls the order in which the items are shown within the list of available items for selection.

Category: Basic Concepts and Terminology

Adding Items to Tables

Context: HL Kit … Basic Concepts and Terminology … Manipulation of Visual Elements … Working With Tables

Contents

Overview

One of the most common mechanisms you'll be using is dynamic tables, as they allow the user to add new items to the character. Whether it be for skills, abilities, powers, spells, gear, or something else, a major part of character creation involves selecting an assortment of options from a list to customize the character. Dynamic tables are the mechanism the Kit uses to accomplish that task.

The Choose Table Vs. Show Table

Dynamic tables are actually two tables in one. First, there is the table in which the selected items are displayed, often called the "show table", and this works as outlined previously. However, there is a second table involved, which is the means through which available items are presented to the user for selection. This is often referred to as the "choose table".

The choose table behaves very similarly to the show table. It uses the same component for filtering the list of items shown. It also uses the List tag expression to filter the items shown. The reason for this is that any item that gets added to the table must fundamentally be shown within the table, so it stands to reason that the same requirements be applied.

The Candidate Tag Expression

However, the choose table also has a **separate** tag expression, which is called the "candidate" tag expression. The Candidate tag expression is distinct because there are often times when you'll want to be more restrictive in deciding which items can be added to the table. Depending on your preferences, the Candidate tag expression can either supersede or be **combined** with the List tag expression to yield the final list of items that are presented to the user for selection.

For example, consider a table of special abilities. Many abilities will be user-selectable, while a number of them will only be added to the character as the result of another choice, such as racial abilities being automatically added only when the corresponding race is chosen. The list of abilities possessed by the character and presented in the show table must include these racial abilities, while the list of abilities that the user can freely choose from must not. To handle this distinction, two separate tag expressions are required.

The Choose Template

In addition to the separate tag expression, the choose table also has a separate template. The list of information users want to view when choosing an item will often be different from the information they want when viewing items that have been added to the table. To accommodate this, a separate template is specified, but there is no reason that the same template cannot be used for both purposes. By decoupling templates from tables, you can readily re-use the same template in multiple places.

Choosing Things Vs. Picks

Another important distinction between the choose table and the show table is the nature of the items presented for selection. Every item that has been added to the character is always a pick, which means that the show table always displays an assortment of picks. However, the choose table can contain either picks **or** things (never a combination). The norm will be things, such as presenting a list of weapons that the user can select from. When things are chosen, a new pick is created and that pick is then added to the character.

Sometimes, though, you will want to display picks that have already been added to the character. For example, the Wizard class in the d20 System must memorize spells from the spellbook. When spells are added to the spellbook, things are chosen and picks are created. However, when spells are memorized, the list of available spells is pulled from the picks that have already been added to the spellbook.

Presenting the Add Option

For consistency, adding an item to a table is always managed through a special option at the bottom of the table. There are two ways to configure the add option. One method is quick and easy, while the other entails more work but provides greater flexibility.

The quick and easy method is to define an "AddItem" script as part of the table. This script specifies the text to be displayed for the add option at the bottom of the table. You can use encoded text to including color highlighting and other formatting, but you are limited to a single line of text. Since a script is involved, you can change the behavior based on the prevailing conditions, such as greying out the text if all available slots have been used up or turning it red to indicate too many slots have been used. If you use this technique, the Kit will automaticallly handle all of the mechanics of presenting the add option.

The alternative is to define a custom template and specify the object to be used with that template. As with everything else for tables, the display of the add option is controlled through a template, and every template must have an object from which is can pull its information. So you can create your own custom template that contains any portals you deem necessary. This technique is rarely necessary, but it does come in handy in some situations. For example, within the d20 System data files, all gear is ascribed a size rating, and each piece of gear must have its size specified when added to the character. So the d20 System data files use a custom template for the add option that includes a menu whereby the size can be specified.

Customizing the Choose Form

In addition to the choose table described above, the form presented for item selection contains other elements. The contents of the choose form can be customized appropriately for whatever is being added. Some of the key elements are discussed below.

Title Bar Contents
Across the top of the choose form is a title. By default, this title contains something generic, along the lines of "Choose an Item from the List Below". While this may be sufficient, it doesn't really tailor the form to a clear purpose. It would be significantly better to even have a title like "Select a Weapon from the List Below". And in situations where the user is selecting items that are limited (e.g. choosing two starting special abilities), it would be even better if the title provided feedback about how many selections remain.

To make this sort of customization possible, dynamic tables possess a "TitleBar" script. The script allows you to construct a suitable title that conveys whatever information is going to be most useful for the user. For example, in Savage Worlds, each starting character is given 15 skill points that can be used to add new skills and improve them. So the Savage Worlds data files display the number of remaining skill points within the title above the choose table for skills. When skill points remain, the title is highlighted in yellow. When they are all used, the title turns grey. And if they are overspent, the title turns red. This type of customization is easy and makes the data files significantly more friendly to use.

Description Region
To the right of the choose table, a large region is provided in which the detailed description of the currently selected item is shown. In some cases, you will find that the default width of this region is too narrow for the items being presented. When this happens, you can use the "descwidth" attribute on the table to change the width of the description region.

You can also control the contents of the description region if you want. This is achieved via the "Description" script for the table. By default, the description region will contain the name and description text for the currently selected item. However, you can customize this content to include more detailed information by defining a suitable Description script.

Buy/Sell Transaction Support
Some types of items involve a transaction for buying and selling the item. For example, weapons and gear will often have a cost associated with them, and characters will have some amount of currency. By integrating the process of buying and selling items directly into the adding and deleting of items, users are able to easily manage their expenses by allowing HL to track everything for them.

For items that entail buy and sell transactions, you can specify suitable templates to be used for each purpose as part of the table. When the choose form is displayed, the buy template is shown in the lower right corner, beneath the description region. The sell template is used whenever an item is deleted from the table.

Category: Basic Concepts and Terminology

Using Choosers

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements ... Working With Tables

You're probably wondering why choosers are being mentioned at this point in the middle of a discussion about tables. The reason is that choosers are basically half of a table and bear discussion now.

Choosers present the selected item in a way that is very similar to menus. However, when the chooser is triggered to select an item, the chooser behaves almost exactly like a dynamic table. In fact, it's no accident that the "choose form" and "choose table" of a dynamic table have been given those names. When a user selects an item from a chooser, he is essentially using the exact same process that is used when adding an item to a dynamic table.

Under the covers, choosers work very much like dynamic tables, and authors configure choosers in very similar ways. Choosers are assigned a component and Candidate tag expression that are used to filter the available options. For customizing the choose form, they also possess TitleBar and Description scripts. Consequently, the definition of a chooser will look in many ways very similar to the definition of a dynamic table.

Category: Basic Concepts and Terminology

Controlling Table Rows and Columns

Multi-Column Tables

Most tables will possess a single column and display as many rows as possible in the space available to them. However, there will be times when you'll want to use tables with multiple columns. For example, if the items being displayed don't need a lot of horizontal space, it may be much better to display them in two or three columns. In the Savage Worlds data files, both Rewards and Resources are perfect candidates for two-column tables.

You can pre-define a table to have multiple columns by setting the "columns" attribute on the table. When a table has multiple columns, the items are sorted so that they increase going downward in the first column, then continue at the top of the second column and increase going downward, then continue at the top of the third column, etc.

Dynamic Control of Columns

You can control the number of columns possessed by a table dynamically from with a Position script. This is accomplished via the "fitcolumns" target reference and is useful when you want to optimize how many columns are displayed based on information you can't know in advance.

The number of columns specified is used as a recommendation when the HL engine triggers the sizing of the table. If a table starts out 400 pixels wide and you specify "fitcolumns" with a value of 2, HL will immediately re-calculate all the sizing details for the table. The width of 400 will be split across two columns, and the template within will have its Position script invoked with an initial width based on the overall width being split.

If you subsequently set the width of the table to 300 pixels, then the previous request for two columns will persist. The table dimensions will be re-calculated, and this time the new width of 300 will be split across two columns. The template will be processed again with the new dimensions.

Dynamic Control of Rows

It's also possible to restrict the number of rows that a table possesses. This can be extremely useful when trying to fit multiple tables into a limit space, and it is accomplished by using the "maxrows" target reference from within a Position script.

Setting the maximum number of rows for a table truncates it to a height that displays no more rows than specified. It is perfectly reasonable for the table to possess fewer rows than specified (e.g. a table with only one item will still only contain and display one item if you set its maximum number of rows to 3).

When fitting multiple tables into a limited space, you can restrict the height of one table to a suitable maximum number of rows. Then you can split the remaining space between the other tables. After positioning the other tables, you can then extend the restricted table to whatever space remains. This way, if the other tables don't use up all the space that you reserved for them, that space can be used by the initially restricted table.

You'll see a number of examples of the above logic used within the starting data files that are provided with the Kit. It's a valuable technique that makes it relatively easy to carve up the available space between multiple tables that can be of wildly varying individual sizes.

Template Sizing Within Tables

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements ... Working With Tables

Contents

Overview

Templates are normally sized either by the layout that contains them or by the template itself. When templates are used within tables, the logic remains the same, but there are additional implications involved.

Triggering of Position Script

The Position script of the template is invoked separately for each individual item being displayed. It is also invoked once when the table sizing process is first initiated. During this initial invocation is when the dimensions of the template are actually used by the containing table. No other positioning logic within the script is applied.

During the initial sizing, you can use the "issizing" target reference to detect that state. When initial sizing is being performed, the template need only calculate its height and can bail out without doing anything else. Since any other script logic is going to be thrown away during the sizing operation, there is no need to perform it.

You will see a check of the "issizing" target reference in most table-based templates within the starting data files provided by the Kit. If sizing is occurring, the script bails out as soon as the dimensions are calculated. While not truly necessary, it is recommended that you use the same technique within your own data files.

Height Control

The height of a template is always expected to be controlled by the template when within a table. Most tables contain fixed-height items, where the height of each item is always the same. For these tables, the template dimensions are expected to be established during the special "sizing" invocation of the Position script.

During sizing, there is **no** specific item available to the template. The HL engine passes in a special, dummy item. This dummy item has values of zero for all numeric fields and values of the empty string ("") for all text-based fields. As a result, the template cannot base its height on the contents of an actual item. It must instead determine its size based on general characteristics, such as font heights and other such information.

Once established during the sizing invocation, the height remains fixed. Any subsequent attempt to set the height to a different value are ignored. The exception to this is with variable-height tables, which are only supported for printed output. Within such tables, each template has its height re-calculated for every item that is output.

Width Control

The handling of the width within table-based templates is significantly more interesting than the height. The width passes through many layers and is treated more as a suggestion than anything else when it is passed into the template. When the sizing invocation of the Position is triggered for a template, the width is initialized to a value that is based on sizing requests made upon the table portal itself within the layout. However, the final decision regarding the width is always at the discretion of the template.

Upon entry to the sizing invocation of the Position script, the width is based on two factors. The total width of the table portal drives the overall width available, while the number of columns specified for the table dictates how the overall width is carved up. Using these two values, the template width is initialized to a value that evenly splits up the overall table width into columns.

In most cases, you will simply accept the width value specified by the table. This is achieved by using the default width that is setup for you and not specifying the width yourself. All of the table-based templates within the starting data files provided with the Kit, and even those within the Savage Worlds data files, simply use the default width given by the table.

If you actually set the template width to a new value, the HL engine honors that new width according to a few important rules.

1. If the table has only one column, the overall table width is changed to be based on the new width given by the template.
2. If the table has multiple columns, the new template width is used to determine how many columns can actually fit within the table, **without** increasing the established table width. Once the number of columns is determined, the table width is changed to be exactly the space needed for those columns. Remember that a table will always be sized to contain at least one item, so setting a template width that is wider than the table width will increase the overall width of the table.

The following are a couple of examples of the above rules being applied. Both examples assume that you start with a table that is 400 pixels in width and that has two columns.

1. If the template sets its width to 300, only one column will now fit. Consequently, the table width is set to 300 pixels to correspond with the one column the table contains.
2. If the template sets its width to 125, three columns with now fit. Consequently, the table width is set to 450 pixels to correspond with the three columns the table contains.

In all cases, the template has final authority over the final dimensions used for the table. The HL engine will suggestion a width that is usually appropriate, but the template is free to override that suggestion as it sees fit.

Category: Basic Concepts and Terminology

Table Headers and Footers

Headers

To help make positioning easier, tables have a built-in option to include a header. Instead of having to position the header separately from the table itself, the Kit merges them into a single unit. When trying to dynamically fit multiple tables into a fixed amount of space, this integration makes the task substantially easier. As the name would imply, headers always appear above the contents of the table.

Headers can be defined for all tables, and they work very similarly to the add option of dynamic tables. The quick and easy way of adding a header is to define a "HeaderTitle" script. The script specifies the text to used for the header, and you can use encoded text to including custom highlighting and formatting, but you may only have a single line of text. Because it's a script, you readily change the contents of the header based on conditions within the character. The Kit will automatically handle all the mechanics of showing the header if you specify a script.

There is a second way to specify a header. You can define a custom template and associate it with the table for use as the header, along with an object from which the template contents can be pulled. This technique is useful when you want to display more complex information within the header. For example, you may want to put column headings above a table, and that entails defining your own header template. You can see a few examples of this within the World of Darkness data files on the Armory tab.

Footers

Only fixed tables can possess a footer, and they always appear beneath the contents of the table. On fixed tables, the footer takes the place of the add option on dynamic tables. The footer can only be defined via use of a template, and that template works just like the header template described above. The only difference is the placement of the footer relative to the table. The object referenced by the footer template is the same one used by the header - it is shared.

Tables Always Auto-Size

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements ... Working With Tables

Tables perform very differently from other portals in terms of their sizing. All portals, including tables, are initially sized at the start of all positioning logic. However, tables will continue to automatically size themselves whenever you make changes to them.

Tables always maintain an integral height and width. This means that the table height is always an exact multiple of the height of the items it contains. It also means that the table width is always an exact multiple of the number of columns it contains. No items are ever partially displayed.

Whenever the size of a table is modified, everything is re-calculated for that table. If the specified dimensions are not an exact multiple of the width and/or height of individual items, then the size of the table is automatically **shrunk** to the next lower integral width and/or height. A table will never be larger than the dimensions assigned to it, but will often be smaller. The lone exception to this that a table will always be big enough to contain a single item. If you attempt to size a table to smaller than is required to display a single item, the table will be sized to show one item.

This auto-sizing behavior is critical to keep in mind when using scripts to optimally fit multiple tables within a layout. The downward adjustment of a table's dimensions by one pixel could result in an entire row or column being dropped from the table. As such, any time that you change the size of a table, you must be sure to retrieve its updated dimensions when positioning another portal relative to it.

Category: Basic Concepts and Terminology

Keying on Items Within Tables

When working with tables, there will be times where you will want to base sizing and positioning on the number of items within a particular table. This will most often occur when trying to intelligently fit multiple tables into a limited amount of space.

The Kit provides two important mechanisms for determining the items in a table. First, there is the "itemcount" target reference that returns the total number of items within the table. Second, there is the "itemsshown" target reference that returns the number of items that are actually visible within the table.

The utility of the item count is probably obvious. By checking the item count, you can determine that a particular table contains no visible items. When generating printed output, a table with no visible items should generally be completely omitted. To achieve this, you can easily check the number of items in a table and set its visibility to zero if the count is zero.

The number of items shown is equally useful. By comparing the number of items shown to another value, you can make appropriate determinations about the what's going on with the table. Most commonly, you will compare the number of items shown against the total number of items in the table. If the values are the same, then you know that the entire contents of the table are visible. When generating printed output, if the two values do not match, then you know that one or more items were not included in the table. This makes it possible to intelligently determine how to lay out the page, since you can quickly determine whether information is omitted and/or will be appearing in a subsequent spillover section of the printout.

Panel Display Order

Context: HL Kit … Basic Concepts and Terminology … Manipulation of Visual Elements

All of the panels shown within HL are assigned a specific order by the data file author. This is accomplished via the "order" attribute on each panel. The attribute specifies a numeric value that indicates the sequence in which the panel should be shown. All panels are sorted based on the order attribute they are assigned.

The resulting sorted order is then used as the sequence in which the panels are actually presented. The tabs across the top of the main window are shown in this order.

Summary panels are always shown after the edit panel, in a separate grouping. They are also sequenced in the order dictated by the attribute each is assigned, but they are grouped separately.

Category: Basic Concepts and Terminology

Using Automatic Placement

Context: HL Kit ... Basic Concepts and Terminology ... Manipulation of Visual Elements

Overview

In an effort to make things as easy as possible, the Kit provides a mechanism called "automatic placement" that makes positioning certain visual elements significantly easier. Although primarily intended for use within sheets, automatic placement can also be used in various places with on-screen visual elements.

Automatic placement can only be used on visual elements within layouts and scenes. Each of these element types manages internal logic to support automatic placement, so you can use the mechanism whenever it suits your needs. Automatic placement assumes that you are placing a progression of visual elements in a vertical arrangement, with each successive element appearing beneath the previous element. Consequently, the mechanism lends itself best to printed output, but it can also come in handy for on-screen positioning.

How It Works

All placement is performed within a rectangular region. Before anything is placed, the bounds of this region are initialized to be the full height and width of the visual container (i.e. the layout or scene). When placement begins, each new placement consumes vertical space within the region. This automatically shrinks the region, moving the top of the region downward to the bottom of each new visual element that is placed.

Automatic placement is performed via the "autoplace" target reference. Each placement can specify a gap that should appear between the new element and the one previously placed, which enables appropriate spacing between visual elements.

Additional target references provide the author with complete control over the bounds of the region within which automatic placement is performed. This makes it possible to place specific elements at the top and/or bottom of the visual container, then adjust the automatic placement region accordingly, and finally perform automatic placement of the remaining visual elements. You can also place elements automatically and then retrieve the bounds of the remaining unused space, after which you can manually place visual elements in that space.

The Rules

In general, automatic placement is very easy to use and very intuitive in how it handles various situations. However, in the interest of clarity, the following specific rules govern how automatic placement behaves.

1. When an element is automatically placed, the width of that element is set to the width of the automatic placement region for the visual container. In other words, each element is sized to take up the full width of the container.
2. When a layout or template is automatically placed, that element is immediately rendered upon completion. This ensures that the sizing of that element is updated so that the top of the region can be accurately moved to the bottom of the element.
3. When automatically placed, most visual elements have their height set to the full remaining height of the automatic placement region. The following caveats apply:
    1. The lone exception to this rule is when a non-table portal is automatically placed within a layout (e.g. a label). In such cases, the height of the portal is assumed to be whatever default height is initialized.
    2. Since the height is set to the full region during automatic placement, it is assumed that every visual element being placed will properly truncate its height as part of its rendering. For example, a template placed within a layout or a layout placed within a sheet must properly set its height at the end of the Position script, basing the height on the extent of the bottommost item within the element.
    3. Table portals automatically determine their extent, so automatic placement of tables works smoothly, without the need for any special handling.
4. When automatic placement attempts to place a visual element that will not fully fit in the remaining space, the region is considered to be fully utilized and no further elements will be placed.
5. If a table is placed and it does not fully fit in the remaining space, as many items as will fit are output. If the table is within a sheet, all remaining items are treated as "spillover" for output in subsequent tables. If within a panel, the table is assumed to provide scrolling to view the excess items.
6. Any visual element that is not displayed at all within the region is designated as non-visible. This means that any table that contains zero items is declared non-visible. Similarly, any templates and/or layouts that do not fully fit are deemed non-visible. IMPORTANT! The key exception to this rule is dynamic and auto tables. Since these tables must allow the user to add items to them at all times, they are always shown even when they contain zero items.
7. Until at least one visual element with actual contents is successfully placed within the region, the gap is always considered to be zero. This ensures that the first item actually placed in the container always starts at the top, regardless of how many elements render no contents.

Category: Basic Concepts and Terminology

Data File Development Process

Context: HL Kit … Basic Concepts and Terminology

Contents

Overview

Before you dive in and start writing your own data files, there are a number of important aspects to the overall development process that you should be familiar with. The topics below strive to relay some basic knowledge that will be incredibly helpful as you begin the authoring process.

Enable Data File Debugging

Before you do anything else, make sure that you've configured HL to enable all of the built-in data file development and debugging aids. By default, HL assumes that users are not creating their own data files, so assorted development facilities within the product are disabled. You need to make sure they are turned on so that you can put them to use.

To enable these tools, go to the "Debug" menu within HL and make sure the "Enable Data File Debugging" option is checked. If it's not checked, click on it once to toggle the state.

Data File Compiler

The HL engine includes a compiler that processes all of the data files you create. The benefits of a compiler are two-fold. First of all, the compiler allows HL to convert all of the disparate data files for a game system into a highly optimized version that can be used. This results in significant performance improvements and much lower memory requirements, thereby allowing HL to manage lots of information efficiently on even older, slower computers.

The second big advantage of a compiler is that the compilation process vets the data files that you've written. If there are errors in the data files, they can be caught in advance and reported to you, allowing you to fix them. Without a compiler, you wouldn't know if you had an error until you tried doing something that uncovered the error, thereby making it harder to verify that your data files work flawlessly.

NOTE! Even though HL uses a compiler, there are some kinds of errors that the compiler simply cannot catch. The vast majority of errors will be caught by the compiler, but some will not. These errors will be trapped and reported as run-time errors, and they are discussed separately in the section on debugging data files.

Compiling Data Files

During the course of developing your data files, there will be times where you want to fully test that everything is working the way you want. There will also be times when you simply want to verify that your changes are valid and compile successfully. You can ask HL to re-compile your data files at any time by going to the "Debug" menu and triggering the "Compile Data Files" option. You'll be prompted to specify the game system to re-compile, after which you'll be shown any error messages that might be encountered during the compilation process.

As long as your files fail to compile, they will not load in the HL, so you should get in the habit of frequently re-compiling your data files. This will uncover problems quickly, since the error must exist in whatever changes you've made since the previous successful compile.

As a convenient shortcut, you can use the <Ctrl-C> key combination to invoke a compile. This makes it easy to regularly verify that your data files are valid at each step along the way as you develop them. Please note that the <Ctrl-C> key combination will **not** work when the input focus is an edit portal, since the <Ctrl-C> is interpreted as a traditional "Copy" command within an edit portal.

Using Quick-Reload

Whenever you make changes to your data files, you'll need to load those changes into HL so that you can use and test them. The obvious way to do this is to go to the "File" menu and select "Switch Game System". However, this approach always shows you the release notes for the game system and potentially the "demo mode" warning, after which you'll be shown the "Configure Hero" form for a new character. After a few dozen times, this process gets really old.

To bypass this, HL includes the "Quick Reload" mechanism, which can be invoked by going to the "Debug" menu and selecting the "Quick Reload" option. This mechanism re-compiles the data files, if necessary, and then reloads them into HL, bypassing the extra steps. It also restores the current tab that is selected. As an added bonus, if you have a saved portfolio loaded, your portfolio is also reloaded. This makes it quick and easy to incrementally modify and test out behaviors associated with selected options.

Take Snapshots Regularly

As you evolve your data files, you will be making significant changes. Even if you are careful, it's likely that you will end up causing everything to break at a few points along the way. When this happens, it can be invaluable to be able to see exactly what has been changed since the last time everything was working fine. In order to do this, you need to have a saved copy of when things were last working. Consequently, we **strongly** encourage you to make a complete copy of your data files at regular intervals, preferably at milestones where everything is working the way you want it. We refer to these copies as "snapshots".

The easiest way to take a snapshot is to use the HLExport tool that is included with the Kit. This tool is designed to package up all of the data files for a game system into a single file that can be easily imported back into HL, and you'll be using this tool to distribute your data files once they're created. In the meantime, though, this tool can also be helpful during development. Using HLExport, you can readily take snapshots of your working data files and save them. If you need to refer back to an old snapshot, you can import the file back into HL and compare the files.

IMPORTANT! If you use HLExport as outlined above and need to reload an old snapshot, be sure to import the files into a **different** directory from the data files you are developing. Otherwise, the old files you import will overwrite your recent changes!

Another simple technique is to make a copy of the entire directory contents for your game system. This allows you to do a direct file-to-file comparison of any file at any time, which can be quite handy at times. The only drawback of this approach is that it often requires more effort than the HLExport technique.

In general, a combination of both techniques will often yield the best results.

Skeleton Data Files

To make it as easy as possible to get started writing data files for a new game system, the Kit includes a starting set of data files. These data files aren't just a hodgepodge of examples, though. They are a fully operational foundation that serves as a framework that you can adapt and build upon to create a solution for virtually any game system.

Since this framework provides all of the basics you'll need and must simply be fleshed out, we refer to this starting point as the Skeleton data files. These data files are minimal in nature, but they offer a solid framework to start with, including a variety of built-in mechanisms that you will likely find yourself using for whatever game system you set out to implement.

Review the Sample Data Files

Before you start trying to develop your own data files, you should first spend a little bit of time familiarizing yourself with the Skeleton data files. Since they will form the starting point for your efforts, you'll benefit substantially by understanding how they work. You should also take the time to review the contents of the data files themselves and get familiar with them, as you'll begin modifying and adapting them.

You can see the Skeleton data files in action by looking at the "Authoring Kit Sample" game system that is installed with HL. If you take a look within the "sample" data file folder, you'll see all of the data files for the game system. Lots of useful insights into the sample data files will be found in the section Exploring the Sample Game System.

The only differences between the Sample game system and the Skeleton data files is that the Sample game system includes an assortment of attributes, skills, abilities, weapons, and whatnot. By including these objects, you can better see how everything works. In contrast, the Skeleton data files omit these objects, since you would otherwise have to delete them before you could begin adding all the material for your own game system.

Functionality, the two sets of data files are identical. The only difference is that the Sample game system includes a variety of things that you can actually play with.

Study the Savage Worlds Example

In addition to the Skeleton data files, a separate separate set of data files is provided that implements a reasonably sophisticated game system - the Savage Worlds system from Great White Games. The Kit documentation includes a detailed walk-through of how the Skeleton data files were adapted for that game system, guiding you through the entire evolutionary process. This offers a concrete example of how to approach your own project.

When starting on the development of data files for a new game system on your own, you'll be starting out with the Skeleton data files and expanding upon them for your own purposes. It would be extremely helpful to first review how the Skeleton data files evolve into the Savage Worlds data files, as there will likely be many similarities with the evolutionary process for your game system. The Savage Worlds Walk-Through will also provide insight into how and when to address different steps in the development process that you can leverage within your own project.

Have a Plan

When you set out to write your own data files, there is one detail that is more important than virtually anything else: have a plan. It is critical that you first do your homework and map out both how everything will work internally and how it will all work and behave. If you launch into writing your data files without a solid plan, you will almost certainly run into a substantial number of surprises and setbacks along the way. Heck, you'll likely have a fair number of those even if you **do** have a good plan in place.

While surprises and setbacks won't stop you from ultimately creating your data files, they will almost certainly cause unnecessary delays and frustration. So you will fare best if you take the time upfront to develop your basic implementation strategy, map out all of the structural pieces you'll need, and design how everything should look and behave for the user.

From a structural perspective, identify all the major elements of the game system. Figure out what types of objects you'll need, then determine the components and compsets necessary. Anticipate the various game behaviors that you need to properly model and map those into the appropriate fields and tags. Sift through all the facets of the game and you'll likely uncover a lot of subtle details that you'll need to fully implement in your data files. It's amazing how many little details lurk within even the most simple game systems. When it comes time to write data files for the game, you'll need to deal with all of them, so it's much better to create a lengthy laundry list upfront so that you can plan for them and not get hit with lots of surprises along the way.

For the visuals, figure out all the various pieces that the user needs to manage. An excellent place to start is the character sheet, but don't stop with that. There are often lots of little details that publishers don't include on the character and that would be a great benefit to support in your data files. Figure out how you're going to visually organize everything across and within the various tab panels. Draw sketches of each tab panel that show what pieces are involved on each. Assemble a road map for how everything will hang together and how the user will interact with it all.

Once you've got the above tasks complete, you're ready to start writing the data files for your game.

Creating the New Game System

When you're ready to launch into developing your own data files, we've made it as easy as possible to get started. There are a vast number of details involved in getting a set of data files to a critical mass where you can successfully load them and experiment with them. There are a similarly large number of mechanisms that will be useful for just about any game system. The good news is that we've saved you the work of having to do all this, as the Kit includes the Skeleton data files that will let you hit the ground running.

Creating the framework for a new game system entails only a few mouse clicks. Go to the "Develop" menu and select the "Create New Game System" option. You'll be prompted to enter both the name of the game system and the name of the folder in which to place the data files. Once that's done, HL will set everything up properly for you. After creation, you can immediately switch to your new game system and set it in place, then you're off and running.

WARNING! There are a number of critical implications with the choice of folder name. Please review the list below before selecting the folder.

- Saved portfolios are associated with a game system via the folder assigned. Changing the folder for a game system after the initial release of data files will result in all existing saved portfolios being rendered inaccessible.
- Until the release of your data files to others, you may freely change the folder name within the definition file, although doing so will invalidate any portfolios you've created thus far.
- Make sure the folder name uniquely identifies the game system. When a user imports the data files, this folder is where the data files will be installed. If two or more game systems use the same folder name, they will overwrite each other.
- The folder name may consist only of alphanumeric characters (i.e. letters and digits). It may not contain any spaces or special characters other than the underscore ('_') and hyphen ('-').

Category: Basic Concepts and Terminology

Advanced Authoring Concepts

Context: HL Kit

This section delves into many of the more advanced concepts involved in creating data files. These concepts cover both structural and visual facets of the data files, as well as suggestions for an effective design philosophy. Simply click on one of the many topics below to learn more about it.

IMPORTANT! This section of the documentation is not yet complete. Topics that have been written will be found at the top of the list below and possess live links. Other topics are simply identified by name, sometimes along with a few notes about what the topic will contain. These topics will be added over time to complete the documentation.

- The "Live" State
- Bootstraps
- Automatically Adding Picks to Actors
- Advanced Script Handling

[TBD]

- Actor Rules
- Required Elements
- Just-in-time Information
  - MouseInfo Scripts
  - Description Scripts
- Game System Logo
- Leveraging Usage Pools [!]
- Advanced Things
  - Uniqueness of Things [!]
  - Pick Linkages [!]
  - Replacement of Things
- Validation
  - Detecting validity of structural elements
  - Reporting validation errors to user
  - Highlighting validation errors to user (i.e. color-coding)
  - Errors Versus Warnings
  - Prompting to select missing information
  - Panel Linkages [!]
- Managing Gear [!]
  - Holders
  - Gear Weight Determination [!]
  - Stackable vs. Non-Stackable [!]
  - Splitting and Merging
- Character Advancement [!]
  - Restricting when advancement is allowed
  - Gizmos with dynamic tagexprs provide configurable advances
  - Advances are displaced to actor
  - Unwind mechanism
  - Autonomous Objects

    Editing of previous advancements
  - Switching back to creation mode

- Pre-Requisites [!]
  - Dependencies on Specific Things [!]
  - Expression-Based Requirements [!]
  - General Pre-Requisites [!]

- Advanced Components and Component Sets [!]
  - Re-using Components in Multiple Component Sets
  - Designing Components for Re-Use
  - Adding "Short Name" Behavior [!]
  - Creation/Deletion Handling

- Advanced Fields [!]
  - Field Types
  - Field Styles
  - Persistence of Fields
  - Bounding of Fields
  - Calculated Fields
  - Finalized Values (Picks vs. Things)
    - Using calculated fields to ensure updates to finalized values
  - Delta for User Fields
  - Formatting of Fields (signed, multiline, integer/float)
  - History Tracking

- Character Sheet Output [!]
  - Standard Sheets
  - Spillover Sheets

- Managing Dossiers

- Statblock Output

- Editor Integration
  - Edit Things
  - Input Things
  - Designing for Editor Integration

- Debugging Techniques

- Distributing Data Files
  - Designing data files for user-extensibility
  - Game System FAQ
  - Appropriate Copyright Information [!]
  - Relying on Minimum Product Features [!]
  - Importance of Release Notes [!]
  - Version Numbers for Data Files [!]
  - Stock Portfolios
  - Using the HLExport Tool [!]
  - Publishing Your Files Through Hero Lab

- Special Tags
  - Actor Tags [!]

    - Identity Tags [!]
- Actor Configuration
    - Sources [!]
        - Required Sources
        - Precluded Sources
    - Rule Sets and Contexts
- Advanced Gizmos [!]
    - Editable Gizmos
    - Associating a Form
    - Setting Up a Default Thing
    - Buy Templates for Gizmos
    - Shadowing [!]
    - Displacement [!]
- Transactions [!]
    - Buy vs. Sell
    - Quantity vs. Cash
    - Using Buy Templates for Customizing Selections (e.g. with choosers)
    - Transaction Tags
- Dashboard
- Tactical Console
    - Initiative
- Design Philosophy
    - Development strategy
    - Designing Data Files
    - Iterative Evolution
- Visual Interactive Behavior
    - Rules for auto-sizing and default sizing of portals
    - Where to use tables, choosers, menus, etc.
    - Showing items within tables
    - Choosing items within tables and choosers
    - Restricting available items within tables and choosers
    - Dual-purpose Headers on Tables [!]
    - Freezing portals
    - Tricks with templates and portals
    - Tricks with layouts, panels, forms, and sheets
    - Automatic placement mechanism for layouts and templates
- Macros
- References
- Condition Tests
- Hidden Things
- Imposing Unwind Logic on Picks [!]
- Minion and Master Relationships [!]

    - Masters Influencing Minions
    - Minions Influencing Masters

- Adaptive Portfolio Loading
    - Load Mods
        - Source Maps
        - Field Maps
        - Silent Objects
    - Load Fixups

- Configuring the Dice Roller [!]

- Evaluation cycle
    - Relative timing of Leads vs. masters vs. minions
    - Rules for sequencing of tasks with the same phase and priority

- Using Visual Resources [!]
    - Built-In Resources
    - Adding Custom Resources
    - Transparent Bitmaps
    - Managing Styles [!]

Categories: Advanced Authoring Concepts | TBD - Not Yet Written

The "Live" State

Context: HL Kit … Advanced Authoring Concepts

Contents

Overview

A variety of objects possess a "live" state. The "live" state controls whether an object **behaves** as if it exists (live) or does not exist (non-live). This makes it possible to automatically include objects within your data files and then dynamically have them appear or disappear, based on conditions within the portfolio that the user is constructing.

The "live" state applies equally to both visual elements and structural elements. A simple example of using the "live" state with visual elements is the various tab panels associated with each different class. All of those tabs are always defined and present, but each appears only when levels of the corresponding class have been added to the character. This is controlled via the "live" state.

For structural elements, the "live" state is primarily used to govern picks. A classic example in the Savage Worlds data files is arcane skills. Arcane skills are only possessed by a character when he has selected the corresponding arcane background. As such, the "live" state is used to control whether the arcane skills are made available.

Tag Expressions Control the State

The "live" state of elements is always controlled via a tag expression. The object upon which the tags are tested will vary between different types of elements, but a tag expression is always employed.

The results of the tag expression dictate the "live" state of the object. If the tag expression returns "true", then the object is considered "live". If the tag expression returns "false", the object is considered "non-live".

The tag expression is evaluated at different points in time, which depend on the nature of the element. Whenever the tag expression is re-evaluated, the behavior of the object can transition if the results of the tag expression change. This means that, whenever the tag expression is re-evaluated, the object can immediately transition from "live" to "non-live", or vice versa. The implications of transitioning the live state are different for visual and structural elements, as outlined in the sections that follow.

Visual Elements

The "live" state of visual elements is always controlled via a Live Tag Expression. This tag expression is always applied against the structural element whose contents are being displayed through the visual element. This is typically an actor, although it can also be a gizmo if the showing a form that edits the contents of the gizmo.

The "live" state is verified when the display is updated, which always occurs after each new evaluation cycle. If the container fails the tag expression test, then the visual element is completely hidden. In addition, any Position script for the visual element is ignored, since there should be nothing to show or position within the visual element.

Structural Elements

The "live" state of structural elements is generally governed by a Container Tag Expression, although other mechanisms also exist. These tag expressions are always tested against the tags on the container of the object, hence the name. The object sets forth the requirements that must be met by any prospective container that wants to hold the object.

The "live" state is tested in two separate situations for structural elements. First of all, when a thing is a candidate for display as a selectable item by the user, the "live" state is checked. If the thing fails the test and ends up being non-live, the thing is completely omitted from the list of items shown to the user. If the this is live, it is shown normally.

Once a thing is added to a container as a pick, the "live" state is again tested as part of every evaluation cycle. When a pick becomes "non-live", it goes dormant. All behaviors associated with the pick are turned off. For example, any scripts that are associated with the pick are simply ignored by the HL engine. The pick is treated as if it were never added in the first place.

However, if a pick becomes "non-live", it cannot simply be automatically omitted from the character. If the user added the pick, then the pick becomes dormant, but some additional behaviors are automatically implemented. First of all, any user-added pick that goes non-live remains visible to the user everywhere. The pick also becomes invalid, which allows for it to be color-highlighted to the user as an error that needs correction. Lastly, a validation error is reported to flag the error to the user.

A structural element that goes non-live has implications upon all elements that are chained to it. This includes all bootstrapped picks, any child gizmo, and any minion that is attached via the pick. Whenever a structural element goes non-live, anything it chains to **also** goes non-live. Consequently, any chained elements go dormant and simply disappear. The only exception to this is a unique pick that is added via multiple roots (e.g. bootstrapped by two or more picks), in which case the chained pick only goes dormant when all of its roots go non-live.

Category: Advanced Authoring Concepts

Bootstraps

Context: HL Kit … Advanced Authoring Concepts

There will be many situations where you want to add a pick to a container automatically. For example, you'll need to make sure that every actor starts out with all of the attributes possessed by characters. You'll also want to re-use abilities, so you'll need to have one thing automatically add another thing.

Within the Kit, this process is referred to as "bootstrapping". Correspondingly, when you add a thing to a container automatically, the resulting pick is referred to as a "bootstrapped" pick. There are a number of important implications associated with bootstrapping, which are discussed in the topics below.

Contents

Things Bootstrapping Other Things

You can bootstrap picks from a variety of situations. However, the most common situation will be when you have one thing bootstrap another. This will regularly occur when the game system has an assortment of abilities that are conferred from a variety of sources. For example, consider the "low-light vision" ability within the d20 System. This ability is conferred by multiple different races, certain magic weapons, etc. You're only going to want to define the ability once, after which you can have each race and weapon simply bootstrap the ability.

When a pick is bootstrapped by another pick, the pick that does the bootstrapping is referred to as the "root" pick. The root pick has complete control control of the bootstrapped pick. If the root pick is deleted, the bootstrapped pick is also deleted. If the root pick goes non-live, so does the bootstrapped pick. The bootstrapped pick still has its own independent existence, but that state is wholly dependent upon the root pick as well.

Situations Where You Can Bootstrap

In addition to things bootstrapping each other, there are a number of additional situations in which bootstrapping of things can be performed. These situations are outlined below. In each of these cases, the dependency relationships outlined above for things bootstrapping things do not apply.

- Things can be bootstrapped onto all actors when the actor is initially created.
- Things can be bootstrapped onto entities as part of the entity's definition. The bootstrapped picks are part of every gizmo created from that entity.
- Things can be bootstrapped onto entities when they are attached. The bootstrapped picks are only included on gizmos created by that thing.
- Things can be bootstrapped onto minions when they are attached. The bootstrapped picks are only included on minions created by that thing.

Bootstrapped Picks are Not Deletable

The topic name pretty much sums it. When a pick is bootstrapped into a container, only the source that performs the bootstrap has control over the pick's existence. Consequently, all bootstrapped picks of fundamentally **not** deletable. Any attempt to delete a bootstrapped pick will fail.

If you want to pre-select a pick into a table or chooser, and you want the user to be able to delete that selection, you need to use a different technique. Please refer to the section "Automatically Adding Picks to Actors".

Bootstrapping the Same Thing Multiple Times

At this point, you may be wondering what happens when the same thing is bootstrapped multiple times into a container. The answer depends on whether the thing is designated as being unique. If not, then separate, independent picks are always created, and their behaviors are not linked in any way. However, if the thing is designated as unique, then only one pick is ever added to a given container. This means that all of the bootstrap actions will actually reference the identical pick.

When the first bootstrap occurs, the new pick is created. Each subsequent bootstrap simply increases the reference count for the pick. As long as at least one of the sources for the bootstrap remains in existence, so will the bootstrapped pick. This is important when things bootstrap over things and those things are user-added. Consider the situation where both ThingA and ThingB bootstrap ThingZ. When the user added ThingA, ThingZ is bootstrapped. When the user adds ThingB, the reference count is increased. If ThingA is deleted, the reference count is decreased, but ThingZ still exists. When ThingB is deleted, the reference count goes to zero and ThingZ is finally deleted as well.

When multiple things bootstrap the same, unique thing, their effects are cumulative upon the new pick. If ThingA specifies an auto-tag for ThingZ, and ThingB specifies a different auto-tag for ThingZ, then both auto-tags are assigned to ThingZ. If only ThingA is added to the container, only its auto-tags are assigned, and the same holds for ThingB. However, if both things are added to the container, both auto-tags are assigned.

If multiple things bootstrap the same, unique thing, all of the conditions associated with the root picks must be handled in accordance with some sort of rules. For example, we'll assume ThingA and ThingB both bootstrap ThingZ, and both ThingA and ThingB have been added to the same container. Now we'll further assume that ThingA has a Live test that is currently failed. This means that ThingA is treated as not existing within the container, although ThingB has no dependency and therefore fully exists. So what happens with ThingZ?

To resolve situations like this, the Kit treats each root pick independently. If **any one** of the root picks for a bootstrapped pick is considered live, then the bootstrapped pick is also considered live. In the example above, this means that ThingZ would be treated as being live due to ThingB being live. The fact that ThingA is not live is irrelevant.

The Mechanics of Bootstrapping

The process of bootstrapping a thing is typically accomplished via the "bootstrap" element. Since there are a variety of places where bootstrapping can be performed, this element is re-used throughout the Kit.

You can also bootstrap things via the "autoadd" element and the "enmasse" element. Both of those elements are exclusively used within structural files.

Component Bootstraps

The vast majority of bootstraps are simple. The bootstrap is always performed for the source context it is defined within. However, bootstraps on components are sometimes an exception. A bootstrap definition on a component will be inherited by all things derived from that component. However, you may only want the bootstrap to be applied to **most** of those things, with some not having the bootstrap.

In these situations, a Match tag expression can be specified with the bootstrap. This tag expression is applied to each **thing** derived from the component. Only the things that satisfy the tag expression are assigned the bootstrap. Note that the tag expression is tested against the tags possessed by each thing, so each thing uniquely controls whether it does or does not receive the bootstrap. This also means that only the initial tags each thing possesses are tested.

IMPORTANT! If a component bootstrap does **not** specify a Match tag expression, the bootstrap is automatically considered a match to all things derived from the component and assigned to all of them.

Conditional Bootstraps

There is another potential wrinkle with bootstraps. Some things can be added to different containers or under different circumstances. Depending on the situation, you may want the bootstrap to be added in some cases but omitted in others. For example, a special ability that is selected directly by the user for an actor may behave one way, while that same ability being added as a power within a magic item may behave a bit differently.

To accommodate special cases like this, a Container tag expression may be specified. This tag expression is applied to the **prospective container** for the new pick. If the container satisfies the condition test, then the bootstrap is added normally. However, if the container does not satisfy the test, no bootstrap is added.

The Condition test on bootstraps works differently from the Match tag expression. In addition to focusing on the container, the test is applied against tags that are dynamically assigned to that container. This means that the Condition test must be scheduled at a specific phase and priority during the evaluation cycle. Any tags that you wish to test for within the container must be handled prior to the timing of the Condition test.

IMPORTANT! All tasks that operate upon a pick must occur **after** any bootstrap Condition tests for the pick. This includes all component scripts. If a task is determined to be scheduled prior to the Condition test, an error will be reported.

IMPORTANT! Within the Condition test, it is valid to test field values, but the source from which these field values are retrieved will vary. The handling of each possible situation is outlined below.

If the bootstrap is being added by another pick, all field value tests are applied to the root pick that is performing the bootstrap.

- If the bootstrap is being added by a gizmo, then all field value tests are applied to the anchor pick that attaches the gizmo.
- If the bootstrap is added by a minion, the field value tests are applied to the master pick that attaches the minion.
- For global bootstraps that are applied to every actor, no field value tests may be utilized, as there is nothing to compare against.
- In all cases, the author is responsible for ensuring that only field values that properly exist are tested, since attempts to access missing fields will result in an error being reported.

Category: Advanced Authoring Concepts

Automatically Adding Picks to Actors

Context: HL Kit … Advanced Authoring Concepts

Contents

Overview

When a new actor is created, it is a virtually empty container. Exactly one thing is automatically added to each actor. It's the "actor" thing that is automatically created for you by the Kit. Other than that, an actor is empty.

So what about all of the various details that are a fundamental part of every actor? Every actor clearly needs to possess all of the basic attributes for the game system (e.g. strength, intelligence, etc.). In most games, the set of skills is pre-set, so those should be part of each actor, too. Every game system has its own unique set of things that should be a basic part of each actor. However, the Kit has no way of knowing what those particular things are for each game system.

To deal with this, the Kit provides three separate mechanisms for specifying which things should be automatically added to every actor. Each mechanism works slightly differently and is intended for use in different situations. Together, they should make it easy for you to pre-configure every actor with all of the picks that it needs. The topics below describe each of these mechanisms.

Adding Individual Things

Each thing that needs to be automatically added to each actor can be explicitly specified. This is accomplished via the "bootstrap" element within a structural file. This element allows you to specify the unique id of the specific thing to be added.

There are quite a few implications associated the bootstrapping a thing into an actor. If you have not yet done so, now would be an excellent time to familiarize yourself with all the particulars of bootstrapping things into containers.

Adding Groups of Related Things

As mentioned above, each game has groups of things that should all be added to each actor. Attributes, skills, saving throws, and other traits are perfect examples. Adding these things individually would be both tedious and error prone. It would also make it more difficult for others to extend the data files by adding their own custom traits, because they would have to manually add any new things they defined.

As long as a group of things can be readily identified by tags they share in common, you have them all automatically added in a single operation. Since all related traits typically derive from the same component, and since every thing possesses a tag for each component it derives from, this requirement is usually a non-issue.

To specify a group of related things that should be added to every actor, use the "enmasse" element within a structural file. This element allows you to specify a tag expression that will identify the related things to be added.

You can also specify tags that will be automatically assigned to each added pick. While rarely needed, this can be useful when the same thing can be also added to the actor additional times by the user.

Pre-Selecting Things Within Tables and Choosers

The above mechanisms allow you to add things to an actor on a fixed basis. This is perfect for attributes and skills, but it doesn't solve all situations. Throughout the interface you design, there will be various tables and choosers where you'll want to pre-select information for the user. In this situations, the user is free to delete or replace the default selection, but one is still provided.

An obvious example is adding an empty character portrait on the Personal tab, since this visually prompts the user to select a portrait of his choice. Another possible situation is when you want to pre-select the contents of a chooser. For example, in the World of Darkness system, the vast majority of characters start out as standard humans, so it's appropriate to pre-select the characters size and speed traits as those of a human. The user is free to change the choices, but he doesn't have to always select them for each new character.

When you come upon a situation where you want to pre-select a thing into a table or chooser, you can use the "autoadd" element within a structural file. This element allows you to specify both the unique id of the thing to be added and the unique id of the portal it is added to.

Any pick that is automatically added to a portal is solely subject to the behaviors for that portal. As such, the pick may not has any Condition tag expression associated with it. In addition, all auto-tags and conditions (e.g. Secondary and Existence) that are normally dictated by the portal are also applied to the pick.

Category: Advanced Authoring Concepts

Advanced Script Handling

Context: HL Kit … Advanced Authoring Concepts

The Kit provides a variety of more specialized control mechanisms for managing scripts. These mechanisms provide the ability to handle special-case situations that may arise when writing data files for some game systems. The topics below cover these various mechanisms.

Contents

Sequencing of Scripts with Identical Timing

When multiple tasks are assigned a common phase and priority, those tasks will be scheduled for evaluation at the exact same time. However, the task scheduler only invokes one task at a time. This means that there would be no guarantee about the order in which two tasks are evaluated that are assigned the same timing. From an authoring standpoint, it's much easier to assign a group of tasks the same phase and priority instead of having to micro-manage which ones occur before each other.

To address this, the Kit provides a set of rules for task scheduling. These rules govern the sequence in which tasks are evaluation that have the same phase and priority. The table below details the order that the Kit uses, with tasks being evaluated in the sequence given, based on their type.

1. Existence tests that are assigned by tables or choosers
2. Bootstrap condition tests that are assigned by root picks
3. Secondary tests that are assigned by tables or choosers
4. Condition tests that are assigned by components
5. Condition tests that are defined explicitly on a thing
6. Calculate scripts defined on any fields
7. Bound scripts defined on any fields
8. Eval scripts that are defined on components or things
9. Evalrule scripts that are defined on components or things
10. Gear scripts that are defined on components

The above rules don't handle a common situation that arises when component scripts are employed. The following topic address how this situation is handled.

Sequencing of Component Scripts

When an eval script or evalrule script is defined for a component, all things derived from that component possess the same script. Since the script is assigned a common phase and priority, all instances of that script will be scheduled for evaluation at the exact same time. As discussed above, the task scheduler only invokes one task at a time, so this means that there is no guarantee about the order in which these scripts are evaluated.

In most cases, there is no need to worry about the respective timing of these scripts, since they don't depend on each other. However, there are situations where it's important that all the scripts being evaluated in a guaranteed order. For example, consider the d20 System data files, where attribute bonuses are chosen every four character levels. It's critical that those bonuses be applied in the exact order that they are selected by the user, since those bonuses have ripple effects elsewhere on the character. Consequently, the Kit imposes rules on the evaluation sequencing of component scripts.

Every component must be assigned a "sequence" attribute. This attribute governs how picks are sequenced to the user by default. It also governs how their tasks will be sequenced during evaluation. Consequently, during task scheduling of eval scripts/rules that are associated with the same component, the corresponding tasks are sorted using the sequence assigned to the component. This ensures that tasks are always scheduled in a consistent fashion. It also extends to the case where things are added multiple times to a container.

In rare situations, you'll need to specify an alternate behavior for task scheduling. To accommodate these cases, individual scripts and rules have an attribute that overrides their behavior. This attribute allows to you specify a different component whose sorting rules must be used when scheduling all tasks for that script.

Limiting Evaluation and Reporting

By default, every eval script/rule is scheduled and processed separately for every pick added to an actor. That behavior is exactly what you'll want 99% of the time, but there are some situations where special handling is needed.

Consider the case where you only want a script to be invoked if a thing is added to the actor. However, what if you also only want the script to be invoked a single time, even if multiples of the thing are added to the actor? There may also be times when you want a rule to be tested for all picks, but you only want the error message to be reported a single time if any of them fail. To deal with these situations, you can specify limits on the evaluation and reporting of scripts and rules.

To limit the evaluation, you can use the "runlimit" attribute to specify the maximum number of times to evaluate the task. This limit is then imposed separately for each container into which the thing is added. If the thing is added ten times to a container, a "runlimit" of one will ensure that the task is only ever invoked a single time. If the thing is added ten times to one container and four times to another container, the task will be invoked once within each container.

To limit the reporting of a rule, you can use the "reportlimit" attribute, which dictates the maximum number of times the rule reports an error. This is critically different from the "runlimit" attribute, which controls the actual invocation. With a report limit, the rule is invoked for every pick, as normal. Only the error message is limited. For example, consider the d20 System data files. If you add multiple class levels to a character, you need to assign hit points for each. You only want to report the error once, but you want all of the picks to be processed so that they will be properly highlighted in red if invalid.

For both evaluation and reporting limits, each thing is normally treated as distinct, with its own limit tracking. Once in awhile, you may want to have all things derived from a given component all contribute to the same limit. In other words, the same limit is imposed whether the user adds the same thing multiple times or different things. This behavior is controlled via the "iseach" attribute on components.

Multiple Tasks with Identical Names

When naming scripts and tag expressions for use with timing dependencies, you can assign the same name to multiple tasks. The first question you're probably asking is why you'd even want to do this, so we'll start there. There will be times when you'll have two or more different scripts that all need to calculate the same field, but they do so for different situations.

A simple example is calculating the net attack value for weapons. Melee weapons base the calculation on a "fight" skill, while ranged weapons base the calculation on a "shoot" skill. As a result, you'll have one script for melee weapons and another for ranged weapons. Both calculate the same field, and they should both occur at the exact same time for consistency, so they should possess the same name. That way, scripts that must occur before or after the net attack value calculation don't have to distinguish between whether it's a melee attack or a ranged attack.

When you assign multiple scripts the same name, only **one** of the scripts is reported in the timing analysis for "errors", "dependencies", and "timing". This is because all identical tasks are assumed to the same for timing purposes, so including them all would be redundant. Identically named tasks **are** still included in the list of all named tasks.

The drawback of only listing one task is that there is normally **no** guarantee which task will be chosen by the Kit for use. In most cases, this isn't a problem, but there is one situation where it is. Consider the case where you have two tasks named "MyTask" and various tasks dependent on those tasks. This will work correctly with no difficulties. However, if you then assign a "before" or "after" dependency to one of those two tasks, you run the risk of having that dependency thrown away by the Kit. If the other task is randomly chosen to be kept from the two named tasks, the dependency will be lost.

In this situation, you could always assign the same dependency to both tasks, but that can be come a real maintenance headache. What you ideally need is a way to ensure that the Kit properly picks the task that has the additional dependency. This is achieved by designating a particular task as the "primary" task from among a group of tasks with the same name. When a task is "primary", the Kit will always choose that task instead of the others with the same name.

Category: Advanced Authoring Concepts

Kit Reference

Context: HL Kit

Contents

Overview

This section details all of the specific formats and mechanisms used within the Kit. This encompasses all of the different file formats, all the scripting contexts and transitions, required and pre-defined elements, and anything else that requires specification. Click on the various topics below to delve into the details for that facet of the Kit.

Revision History

The Authoring Kit is an evolving toolset. A detailed summary of the changes and enhancements made within each new version after the initial V3.0 release is accessible via the links below.

- Functionality Changes and Enhancements
- Skeleton Data File Changes and Enhancements

XML Details

Most of the basics regarding the Kit's use of XML files and their implications can be found in the section on XML Files. Additional reference details are outlined in the topics below.

- XML Character Encoding Set

Conventions Used Below

The reference section of this documentation utilizes a variety of notational conventions for presenting how things work. This includes the syntax used for data files, as well as the formats for other types of files, which are outlined in the topics below.

- XML Attributes in Data Files
- Specifying PCDATA in Data Files
- Optional Attributes in Data Files

Tags and Tag Expressions

Tags are a fundamental building block that a wide range of mechanisms leverage through the Kit. Tags are utilized to identify and classify objects through tag expressions. The following topics delve into the various facets of using tags.

- Leveraging Tags Via Tag Expressions
- Tag Expression Types

Scripting Language

The types of behaviors that exist within the realm of RPGs is limitless. As such, it's impossible for HL to anticipate everything, so the Kit provides a versatile scripting language that enables data files to adapt to virtually any game system. The scripting language has many facets that you should be familiar with, and the topics below outline the information you'll need.

- Scripting Language Overview
- Language Syntax
- Declaring Variables
- Basic Language Mechanisms
- Flow Control
- Other Language Statements
- Language Intrinsics
- Special Symbols
- Script Macros
- Re-usable Procedures
- Debugging Mechanisms
- Compiler Error Messages

Script Data Access

The majority of your scripts will focus on accessing and manipulating the data managed within HL. This will involving identifying both the structural and visual elements throughout the data hierarchy. The following topics detail the various pieces involved in data access.

IMPORTANT! Be sure you are familiar with the basics of data manipulation before reviewing this content.

- Script Contexts
- Context Transitions
- Target References
- Data Access Examples
- Employing Script Macros
- Script Types

Structure of Data Files

There are a number of different types of files that comprise the data files for a game system. Each of the topics below describes the structure of one of these file types.

- Definition File Reference
- Structural File Reference
- Data File Reference

Auto-Defined Elements

The Kit automatically defines a variety of different structural and visual elements for use in common situations. These auto-defined elements help to streamline and simplify the authoring process, and they are detailed in the topics below.

- Pseudo-Fields
- Auto-Defined Tags and Tag Groups
- Auto-Defined Components and Fields
    - journal, transact, stackable, gear, shortname, etc.
- Auto-Defined Component Sets
- Auto-Defined Things
- Auto-Defined Sort Sets
- Built-in Resources
- System Resources

Required Elements

There are an assortment of structural and visual elements that every set of data files is required to define. By standardizing on a core set of objects, everything becomes simpler to manage for the data file author. To make the process as easy as possible, the Skeleton data files pre-define these necessary pieces, which you can leave as is or modify if you wish.

- Required Panels
- Required Forms
- Required Components and Fields
- Required Component Sets
- Required Things

Other File Formats

In addition to the various data files, HL utilizes a few other types of files. The contents of these files is documented in the topics below.

- Timing Report File Reference
- Portfolio File Reference

Category: Kit Reference

Functionality Revision History

Context: HL Kit … Kit Reference

Beginning with the V3.1 release, a summary of the functional changes and enhancements to the Kit within each release is provided below.

- V3.1 Release
- V3.2 Release

V3.1 Revision History

The following changes and additions were introduced in V3.1.

1. Extended "foreach" statement to support "foreach thing in component" syntax, which processes all things derived from a specified component.
2. Extended "foreach" statement to support "foreach bootstrap in thing", which processes all things that are directly bootstrapped by an identified thing.
3. Extended "foreach" statement to support "foreach bootstrap in entity" for use within Description scripts, which processes all things directly attached to the entity associated with the current thing context.
4. The "isentity" target reference on picks and things indicates whether the item has a child entity attached.
5. The "isgizmo" target reference behaves equally for things as well as picks.
6. Picks with attached entities are treated the same as minions with respect to the scheduling of the final condition test. This means the final condition test is scheduled at the time of the latest condition test instead of at the earliest script. This change ensures that a single condition test can be defined on a component so that all child picks can safely test their live state, which depends on the live state of the gizmo, which depends on the live state of the pick that attaches the gizmo.
7. When auto-tags are assigned via a bootstrap with a condition test, the tags are only assigned to the pick when the condition test is satisfied.
8. Bootstraps can selectively override the values assigned to fields within the bootstrapped pick via the "assignval" child element.
9. The "editthing" element can be assigned a "prefix" attribute, which will be used by the Editor to setup an initial unique id for new things created within the Editor.
10. History tracking for fields with "stack" behavior supports the "=" operator, which overwrites the current value with the new value specified.
11. If a field history entry adds or subtracts a negative value, the operator is automatically inverted and the value is negated, changing entries from "+-3" to "-3".
12. The "history" target reference for fields and values supports an optional starting value to be shown in the report. Also, the splicing text can be omitted, in which case ", " is used.
13. The "history" attribute for fields has a "changes" option available that omits any adjustments that yield no actual change, such as "+0" or "*1".
14. When "best" history tracking is used for fields, any adjustments that fail to apply a meaningful change are ignored (like the "changes" behavior above).
15. Portals of type "table_fixed" are automatically omitted by the auto-place mechanism when empty.
16. Field history tracking is now allowed for fields that possess a Finalize script.
17. The "output_dots" portal type was added to make it easy to insert a series of dots between two portals within character sheet output.
18. Header portals used within dual-purpose templates may now use scripts, although the initial script context is undefined and the author must immediately transition to a valid context.
19. The "ischanged" target reference on fields and values returns whether the field value has been changed in some way from its original starting state.
20. The "pushtags" and "pulltags" target references now support containers as either the source or destination, or both.
21. The pseudo-field "thingname" provides access to the original name assigned to a thing, which was otherwise inaccessible if the name of a pick was either modified via a script or renamed by the user.
22. The "scenevalue" target reference provides a mechanism for managing "global" values within the context of a scene to allow communication between the scene and all layout and template scripts within it.
23. The global "state" script context provides support for setting and getting persistent values that are global and exist at the

portfolio level, outside of any actor. This mechanism makes it possible to manage state across the entire portfolio.

24. The management of global, persistent, named sets of values is provided within the "global" script context via the "setrandom", "setextract", "setremain", and "setdiscard" target references.
25. The NewCombat and NewTurn scripts possess an "@isfirst" special symbol, which indicates whether the script is being invoked on the very first actor. This allows one-time handling at the start of a combat or turn that spans all actors.
26. The NewCombat and NewTurn scripts are always invoked *before* the Initiative script is invoked for any actor.
27. The "shortname" field is now synthesized at a priority of 100 within the "Render" phase, if that phase exists, else in the last phase of evaluation.
28. The minimum font size supported by "sizetofit" is now 6-point when rendering to the screen and 4-point when rendering onto sheets for printouts.
29. The primary and secondary initiative values for each actor will persist if they are not set to a new value within the Initiative script.
30. The InitFinalize script can be specified in the definition file and will be used as the Finalize script for the "initiative" field.
31. The "initminimum" and "initmaximum" attributes were added to the "behavior" element in the definition file, dictating the lower and upper bounds for the initiative value when the user adjusts the value via the incrementer in the TacCon.
32. The "gaphorz" and "gapvert" target references on visual elements have been renamed to "gapx" and "gapy", respectively, to eliminate confusion regarding their behavior.
33. The various "gap" attributes on table portals have been renamed to eliminate confusion about their use. The "showgaphorz" and "showgapvert" attributes are now "showgapx" and "showgapy", respectively. The "choosegaphorz" and "choosegapvert" were renamed to "choosegapx" and "choosegapy".
34. Printing of a spillover page continues until a page is generated that contains no data to output, at which point printing continues with the next page.
35. Encoded text supports the indenting of the first line of a paragraph via the "{indent value}" syntax. Both normal and hanging indents are supported.
36. Menu styles possess the "droplist" and "droplistoff" attributes, allowing the author to override the bitmaps used for the drop-arrow.
37. The LeadSummary script is allowed to call procedures, which must be either the "container" context or "any" script type.
38. Picks support the "uniqindex" target reference, which returns a value that uniquely identifies the pick throughout the entire portfolio.
39. Only "derived" fields may utilize a Calculate script.
40. The "output_separator" portal can be used within sheet output to insert an solid black line within the dimensions given.
41. The "isroot" target reference (of picks) is accessible from the "template" script context.
42. If the "@message" special symbol is set within an Eval Rule, but the "@summary" symbol is not, the returned message text is automatically used as the summary for display on the validation summary bar.
43. The "exprreq" and "pickreq" elements on things support the "onlyonce" and "issilent" attributes, exactly the same way as they work on standard "prereq" elements.
44. The "image_literal" portal supports the "isbuiltin" attribute, which identifies a bitmap that is provided by HL for general use.

V3.2 Revision History

The following changes and additions were introduced in V3.2.

1. The dossier being output adds a "dossier.<id>" global tag to the portfolio during output. This allows scripts to check which dossier is being output.
2. An "EndCombat" script is invoked on each actor when combat ends.
3. An "endcombat" procedure type is supported for use with the EndCombat script.
4. The "meta" text encoding allows the alignment behavior of bitmaps to be controlled relative to the position of text when mixing text and bitmaps.
5. The scripting language supports the "doneif" statement to simplify coding.
6. The "state" script context supports the "thing[id]" transition to directly access all facets of a thing.
7. The "modify" target reference for fields supports the "#" operator, which suppresses the display of any operator within the resulting history report.
8. The "modify" target reference for fields supports the "$" operator, which inserts a history entry with no field value and only a

text description.

9. Things possess the "holdlimit" tag expression to restrict the types of things that can be held by the thing within the gear containment hierarchy. This makes it possible to assign laser sights and such to weapons.
10. The "it_field" input thing has an optional boolean "multiline" attribute to support the entry of multi-line text fields within the Editor.
11. Thing-based menus can specify an alternate field to display for the menu item via the "namefield" attribute.
12. The "heromatch" target reference on picks and containers behaves the same as "tagmatch", except that the initial context is always assumed to be the current actor. This allows arbitrary matches against the hero from anywhere.
13. Sources possess both the "maxchoices" and "minchoices" attributes. These allow an author to specify a minimum and/or maximum number of child sources that can be selected by the user.
14. Added the "plaintext" intrinsic function to the scripting language.
15. Added the "amendthing" target reference to the "thing" script context, which allows pertinent feats in 4E to directly modify the powers that they impact.

Category: Kit Reference

Skeleton Data File Revision History

Context: HL Kit … Kit Reference

The Skeleton Data Files were initially released in V3.0 and have evolved since that time. If you started work on your data files with an older release than is currently available, you should probably integrate the changes listed below for each subsequent release.

- V3.1 Release
- V3.2 Release

V3.1 Revision History

IMPORTANT! A detailed list of the specific changes made to each file can be found here.

The following changes and additions were introduced in V3.1.

1. Added highlighting of any currently equipped weapon and/or armor on the Armory summary panel.
2. Eliminated use of the "mousepos" attribute within MouseInfo scripts whenever it specified the default behavior was to be used.
3. Fixed a bug in the standard character sheet that failed to handle weapons that are non-stackable.
4. Integrated use of the "output_dots" portal within the sample character sheet framework that is provided.
5. Added a few lines of missing code to Position script for the "oAdjPick" template.
6. Increased the "maxfinal" length of "grStkName" field within "Gear" component to 100.
7. Fixed problem with the color highlighting not showing red properly in Finalize script for "resAddItem" field of "Resource" component.
8. Revised the gear template to be more intelligent and adaptive.
9. Eliminated some extraneous code from the "power" portal within the "dashboard" template.
10. Fixed a problem in the "DshActive" procedure that was not properly outputting activated abilities that weren't in-play adjustments.
11. Added the "lblSmlLeft" style for general use.
12. Utilized the "lblSmlLeft" style within the TacCon.
13. Fixed problems with the Eval script for generating the name within the "Adjustment" component.
14. Corrected the behavior of the Label script for the "summary" portal within the "tacPick" template of the Tactical Console.
15. Cleaned up the code in various scripts within the "tacPick" template of the Tactical Console.
16. Renamed the "damage" and "status" portals within the "tacPick" template of the Tactical Console to "status1" and "status2", respectively.
17. Renamed the "column1" portal within the "tacPick" template of the Tactical Console to "traits".
18. Renamed the "DashTacCon.Column1" tag to "Traits".
19. The contents of the "peace" and "summary" portals within the "tacPick" template of the Tactical Console are sorted by name for display.
20. The "summary" portal uses "sizetofit" to shrink a bit whenever its contents are too large to fit in the available space.
21. The "static" form include built-in handling for accessing the master when a minion is being edited.
22. Defined the "actMaster" style and added the corresponding bitmaps as built-in resources.
23. The "Domain" component automatically integrates the domain into the name of the pick, plus the "shortname" field if one exists.
24. The form for advancements uses the base "thing" name when synthesizing a name for display instead of the normal name, providing full control over how the domain is shown for advancements.
25. Suitable abbreviations are now defined for all traits.
26. Revised the logic for shrinking text on the advancements form.
27. Revised the logic for limiting line height on advancements form.
28. Eliminated use of "textheight" for validation report sizing in the character sheet.
29. Revised a number of Position scripts within the character sheet to make proper use of "sizetofit" (some original uses were ineffectual).
30. Fixed problem where the Journal tab was showing the total XP based on the contents of the usage pool instead of the "resXP"

resource.

31. Heroes track their "dead" state, which is shown on the TacCon.
32. Added pre-defined "menuSmall" style for smaller menus.
33. Added built-in support for handling things that require special user-selection behaviors, including checkboxes, array-based menus, and thing-based menus. This includes the "UserSelect" component for behaviors and "UserSelect" template to handle visuals.
34. Added built-in thing for handling natural armor, which requires the "defDefense" field to be "derived" instead of "static".
35. Added handling to the "SimpleItem" template that automatically changes the color of any non-deletable items to indicate that state.
36. If an equippable item possesses the "Equipment.Natural" tag at the time of creation, the item is initialized to the equipped state but may thereafter be changed.
37. Added option for centering the name within the "SimpleItem" and "LargeItem" templates.
38. Added the "Equipment.StartEquip" tag and handling for it within the Creation script of the "Equippable" component.
39. Revised the validation rules within the "Domain" component so that any panel linked to the thing is properly highlighted when the domain is required and not specified.
40. The "Domain" component supports customization of the term shown to the user for the domain within validation errors via assignment of a tag from the "DomainTerm" tag group.
41. The advancements mechanism supports the "DomainTerm" behaviors when displaying options that leverage domains.
42. Moved the user manual into the "docs" folder beneath where the data files are stored to keep it separate from the other data files.
43. Abilities marked as "creation only" now verify the appropriate state via the "state.iscreate" test.
44. Revised timing of setting the delta for the "trtUser" field in the file "traits.str".
45. Revised timing of the scripts that tally attribute and skill points within "traits.str".
46. Added a "notation" advance to the advancements mechanism.
47. Sheet output of abilities must be limited to a single line high.
48. Changed the timing of the script handling auto-equipped gear, as it was scheduled much earlier than necessary and limiting authoring options.
49. Moved a variety of bitmaps into the "builtin" folder, which then required changing most "image_literal" portals to utilize the bitmaps there.
50. Added an assortment of named color resources that are then used within the various styles, which allows for easier re-use and replacement by an author.
51. The Finalize script of the "acPPSumm" field on the "Actor" component must shown the "trkUser" field instead of "trtLeft".
52. Revamped the starting HTML file for use as a User Manual.
53. The ranged weapons table "arRange" must reference the "Gear" component so that the appropriate transaction scripts are invoked.

V3.2 Revision History

IMPORTANT! A detailed list of the specific changes made to each file can be found here.

The following changes and additions were introduced in V3.2.

1. Fixed bug where character sheet output that continued onto subsequent pages did not properly position the right-side column of output.
2. Added use of the "prefix" attribute on "editthing" elements within the Editor.
3. Unspent resources report validation warnings and highlight the appropriate tabs unless assigned the "Helper.NoMinimum" tag.
4. Fixed bug where the "lot cost" of a piece of gear was being calculated incorrectly.
5. Added new system resources that authors can override when tailoring the interface for a custom game system.
6. Fixed bug where ammunition was not having its quantity setup properly for tracking on the In-Play tab.
7. Eliminated the edit portal length from three edit portals that exceeded the maximum allowed length of the underlying field.
8. Fixed problem with the handling of abilities, where they were not being properly flagged on the actor for indication to the user.
9. Added support for entering "Ammunition" via the Editor.

10. Added support for the "Lot Cost" and "Weight" fields for all types of equipment within the Editor.
11. Added a shell "editor.htm" file that can be used as the basis for providing suitable documentation for adding content to the game system via the Editor.

Category: Kit Reference

Skeleton File Changes V3.1

Context: HL Kit … Kit Reference … Skeleton Data File Revision History

Contents

File: "definition.def"

Line 24: Eliminate the "manualroot" attribute so that the manual location uses the default.

Line 46: Replace the "required" attribute with "3.1" instead of "3.0".

File: "tags.1st"

Line 52: Insert the new tag shown below.

```
<value id="NoAutoName"/>
```

Line 65: Insert the new tag shown below.

```
<value id="Dead"/>
```

Line 79: Rename the "Column1" tag to "Traits".

Line 90: Insert the new tag shown below.

```
<value id="StartEquip" name="Gear starts out equipped"/>
```

Line 245: Insert the new tag group definition shown below.

```
<group
  id="DomainTerm"
  dynamic="yes">
  <value id="Domain"/>
  </group>
```

Line 254: Insert the new tag shown below.

```
<value id="CenterName"/>
```

Line 256: Insert the new tag group definitions shown below.

```
<group
  id="ChooseSrc1"
  visible="no">
  <value id="Thing" name="All Things"/>
  <value id="Container" name="All Picks on Container"/>
  <value id="Hero" name="All Picks on Hero"/>
  </group>
<group
  id="ChooseSrc2"
  visible="no">
  <value id="Thing" name="All Things"/>
  <value id="Container" name="All Picks on Container"/>
  <value id="Hero" name="All Picks on Hero"/>
  </group>
```

File: "advancement.core"
Line 19: Added the "Notation" tag shown below.

```
<value id="Notation"/>
```

Line 160: Insert new logic to handle notation advancements at the start of the Eval script, as shown below.

```
~if this advancement has an annotation, there is no user-selection, so build
~the name from our pieces and we're done
if (tagis[Advance.Notation] <> 0) then
  perform gizmo.child[advDetails].setfocus
  field[livename].text = field[name].text & ": " & focus.field[advUser].text
  done
  endif
```

File: "components.core"
Line 273: Change the reference to the "component.Ability" tag to "component.shortname".

Line 278: Change the reference to the "component.Ability" tag to "component.shortname".

Line 283: Insert the code below to use the standard name if no short name is defined.

```
~if we don't have a short name, just use the regular name
if (empty(short) <> 0) then
  short = name
  endif
```

Line 337: Insert the new "UserSelect" component definition that is provided below.

```
<component
  id="UserSelect"
  name="User Selection">

  <!-- Text to display with the checkbox
        NOTE! If this field is empty, it means NO checkbox is shown for the pick.
  -->
  <field
    id="usrChkText"
    name="Checkbox Text"
    type="derived"
    maxlength="100">
    </field>

  <!-- Indicates whether the checkbox is selected is not -->
  <field
    id="usrIsCheck"
    name="Checked?"
    type="user"
    minvalue="0"
    maxvalue="1">
    </field>

  <!--  Label associated with the first thing-based menu -->
  <field
    id="usrLabel1"
    name="Thing Menu Label #1"
    type="static"
    maxlength="50">
    </field>

  <!-- Tracks the first selection made when a menu choice is required -->
  <field
    id="usrChosen1"
    name="Chosen Thing / Pick #1"
    type="user"
    style="menu">
    </field>

  <!-- Candidate tagexpr used to determine which picks/things are shown in menu #1
        NOTE! If this field is empty, it means NO first menu is shown for the pick.
  -->
  <field
    id="usrCandid1"
    name="Candidate TagExpr for Menu #1"
    type="derived"
    maxlength="1000"
    defvalue="">
    </field>

  <!-- Source to pull choices from within menu #1 -->
  <field
    id="usrSource1"
    name="Source for Menu #1 Choices"
    type="derived">
    <!-- Determine the source to choose from based on the tag, defaulting to "Hero" -->
    <calculate phase="Render" priority="10000"><![CDATA[
      if (tagis[ChooseSrc1.Thing] <> 0) then
        @value = 0
      elseif (tagis[ChooseSrc1.Container] <> 0) then
        @value = 1
      elseif (tagis[ChooseSrc1.Hero] <> 0) then
        @value = 2
      else
        @value = 2
        endif
      ]]></calculate>
    </field>

  <!--  Label associated with the second thing-based menu -->
  <field
    id="usrLabel2"
    name="Thing Menu Label #2"
    type="static"
    maxlength="50">
    </field>

  <!-- Tracks the second selection made when a menu choice is required -->
  <field
    id="usrChosen2"
    name="Chosen Thing / Pick #2"
    type="user"
    style="menu">
    </field>

  <!-- Candidate tagexpr used to determine which picks/things are shown in menu #2
        NOTE! If this field is empty, it means NO second menu is shown for the pick.
  -->
  <field
    id="usrCandid2"
    name="Candidate TagExpr for Menu #2"
    type="derived"
    maxlength="1000"
    defvalue="">
    </field>

  <!-- Source to pull choices from within menu #2 -->
  <field
```

```
    id="usrSource2"
    name="Source for Menu #2 Choices"
    type="derived">
    <!-- Determine the source to choose from based on the tag, defaulting to "Hero" -->
    <calculate phase="Render" priority="10000"><![CDATA[
      if (tagis[ChooseSrc2.Thing] <> 0) then
        @value = 0
      elseif (tagis[ChooseSrc2.Container] <> 0) then
        @value = 1
      elseif (tagis[ChooseSrc2.Hero] <> 0) then
        @value = 2
      else
        @value = 2
        endif
      ]]></calculate>
    </field>

  <!--  Label associated with the array-based menu -->
  <field
    id="usrLabelAr"
    name="Array Menu Label"
    type="static"
    maxlength="50">
    </field>

  <!-- Array of text items user can select from
        NOTE! If the 0th element is empty, it means NO menu is shown for the pick.
  -->
  <field
    id="usrArray"
    name="Array of Items to Choose"
    type="derived"
    style="array"
    arrayrows="10"
    maxlength="30">
    </field>

  <!-- Item selected from the array -->
  <field
    id="usrSelect"
    name="Selected Item in Array"
    type="user"
    maxlength="30">
    </field>

  <!-- Initialize the current array-based selection from the array if not defined -->
  <creation><![CDATA[
    if (empty(field[usrArray].arraytext[0]) = 0) then
      if (field[usrSelect].isempty <> 0) then
        field[usrSelect].text = field[usrArray].arraytext[0]
        endif
      endif
    ]]></creation>

  <!-- Integrate the various user selections into the name of the pick
        NOTE! Must be scheduled after the "shortname" field is synthesized at Render/100.
  -->
  <eval index="1" phase="Render" priority="500"><![CDATA[
    ~if we're not supposed to auto-amend the name for this pick, we're done
    if (tagis[User.NoAutoName] <> 0) then
      done
      endif

    ~if we have thing-based menus, determine the text to append to the name
    var choices as string
    if (field[usrCandid1].isempty = 0) then
      if (field[usrChosen1].ischosen <> 0) then
        choices = field[usrChosen1].chosen.field[name].text
      else
        choices = "-Choose-"
        endif
      if (field[usrChosen2].ischosen <> 0) then
        choices &= ", " & field[usrChosen2].chosen.field[name].text
        endif

    ~if we have an array-based menu, determine the text to append
    elseif (empty(field[usrArray].arraytext[0]) = 0) then
      choices = field[usrSelect].text

    ~if we have a selected checkbox, determine the text to append
    elseif (field[usrChkText].isempty = 0) then
```

```
      if (field[usrIsCheck].value <> 0) then
        choices = field[usrChkText].text
        endif
      endif

    ~if we have no text to append, we're done
    if (empty(choices) <> 0) then
      done
      endif

    ~add the selection to both the livename and shortname (if present) fields
    field[livename].text = field[name].text & ": " & choices
    if (tagis[component.shortname] <> 0) then
      field[shortname].text &= " (" & choices & ")"
      endif
    ]]></eval>

  <!-- Report a validation error if no selection has been made for a menu selection -->
  <evalrule phase="Validate" priority="10000" message="You must choose an option"
summary="Choose!"><![CDATA[
    ~determine the number of menus that NEED selection
    ~Note: Remember that a non-empty tagexpr field indicates menu selection is used.
    var needed as number
    needed = !field[usrCandid1].isempty + !field[usrCandid2].isempty
    needed += !empty(field[usrArray].arraytext[0])

    ~determine the number of menus that HAVE selections
    var actual as number
    if (field[usrCandid1].isempty = 0) then
      actual += field[usrChosen1].ischosen
      endif
    if (field[usrCandid2].isempty = 0) then
      actual += field[usrChosen2].ischosen
      endif
    if (field[usrSelect].isempty = 0) then
      actual += 1
      endif

    ~if the user has chosen something whenever required, we're valid
    if (actual >= needed) then
      @valid = 1
      done
      endif

    ~mark any associated tab as invalid
    if (ispanel <> 0) then
      linkvalid = 0
      endif
    ]]></evalrule>

  </component>
```

Line 400: Add the new Eval script below to the "Domain" component.

```
<eval index="1" phase="Render" priority="500"><![CDATA[
  ~if we don't need a domain, there's nothing to do
  if (tagis[User.NeedDomain] = 0) then
    done
    endif

  ~if we're not supposed to auto-amend the name for this pick, we're done
  if (tagis[User.NoAutoName] <> 0) then
    done
    endif

  ~if we don't have a domain, use a placeholder for it
  var domain as string
  if (field[domDomain].isempty = 0) then
    domain = field[domDomain].text
  else
    domain = "????"
    endif

  ~add the domain to both the livename and shortname (if present) fields
  field[livename].text = field[name].text & ": " & domain
  if (tagis[component.shortname] <> 0) then
    field[shortname].text &= " (" & domain & ")"
    endif
  ]]></eval>
```

Line 401: Replace the original Eval script with the new one shown below.

```
<evalrule index="1" phase="Validate" priority="9000" message="????"><![CDATA[
  ~if no domain is needed or the domain is specified, we're valid
  if (tagis[User.NeedDomain] = 0) then
    @valid = 1
  elseif (field[domDomain].isempty = 0) then
    @valid = 1
    endif

  ~if we're valid, get out of here
  if (@valid <> 0) then
    done
    endif

  ~if we have a linked panel, flag it as invalid
  if (ispanel <> 0) then
    linkvalid = 0
    endif

  ~synthesize an appropriate message using the correct domain term
  var term as string
  term = tagnames[DomainTerm.?]
  if (empty(term) <> 0) then
    term = "Domain"
    endif
  @message = term & " must be specified"
  ]]></evalrule>
```

File: "actor.str"
Line 163: Change the reference to the "trtLeft" field to "trtUser".

Line 221: Replace the Eval script with the revised script below.

```
<eval index="1" phase="Final" priority="1000"><![CDATA[
  ~if no damage has been incurred, assign a tag to indicate that state
  if (field[acHPNow].value >= field[acHPMax].value) then
    perform hero.assign[Hero.NoDamage]

  ~if the hero is dead or otherwise out of combat, indicate that state
  elseif (field[acHPNow].value = 0) then
    perform hero.assign[Hero.Dead]
    endif
  ]]></eval>
```

File: "equipment.str"
Line 27: Increase the "maxfinal" value from "50" to "100".

Line 219: Insert the Creation script shown below.

```
<creation><![CDATA[
  ~if this is natural equipment, initialize the equipped state
  if (tagis[Equipment.Natural] <> 0) then
    field[grIsEquip].value = 1
    endif

  ~if this equipment is supposed to start out as equipped, initialize the state
  if (tagis[Equipment.StartEquip] <> 0) then
    field[grIsEquip].value = 1
    endif
  ]]></creation>
```

Line 229: Change the Eval script priority from "1000" to "4000".

Line 584: Change the field type from "static" to "derived".

File: "miscellaneous.str"
Line 103: Insert the new lines of code below into the Finalize script.

```
  @text = "{text ff0000}"
elseif (unspent > 0) then
```

File: "traits.str"

Line 50: Change the timing of the Bound script, assign it a name, and establish a "before" dependency, as shown below.

```
<bound phase="Traits" priority="1000" name="Bound trtUser">
  <before name="Calc trtFinal"/><![CDATA[
  @minimum = field[trtMinimum].value
  @maximum = field[trtMaximum].value
  ]]></bound>
```

Line 97: Change the timing of the Eval script from "9999999" to "5000".

Line 130: Change the timing of the Eval script and setup "before" and "after" dependencies, as shown below.

```
<eval index="2" phase="Traits" priority="10000">
  <before name="Calc resLeft"/>
  <after name="Bound trtUser"/><![CDATA[
  hero.child[resCP].field[resSpent].value += (field[trtUser].value - 1) * 7
  ]]></eval>
```

Line 175: Change the timing of the Eval script and setup "before" and "after" dependencies, as shown below.

```
<eval index="2" phase="Traits" priority="10000">
  <before name="Calc resLeft"/>
  <after name="Bound trtUser"/><![CDATA[
```

Line 253: Change the test from keying on the usage pool to the actual mode, as shown below.

```
~if the mode is creation, we're valid
if (state.iscreate <> 0) then
  @valid = 1
  done
  endif
```

File: "styles_output.aug"
Line 217: Insert the new style shown below.

```
<style
  id="outDots">
  <style_output
    textcolor="202020"
    font="ofntnormal"
    alignment="left">
    </style_output>
  </style>
```

File: "styles_ui.aug"
Line 236: Insert the new resource definition shown below.

```
<resource
  id="fntmenusm">
  <font
    face="Arial"
    size="30"
    style="bold">
    </font>
  </resource>
```

Line 268: Insert the new color resource definitions shown below.

```
<!-- color used for normal text throughout the ui -->
<resource
  id="clrnormal">
  <color
    color="f0f0f0">
    </color>
  </resource>

<!-- color used for text on the static panel - not quite as bright -->
<resource
  id="clrstatic">
  <color
```

```
    color="d2d2d2">
    </color>
  </resource>

<!-- color used for text in title labels -->
<resource
  id="clrtitle">
  <color
    color="c0c0c0">
    </color>
  </resource>

<!-- color used for names of automatically added picks -->
<resource
  id="clrauto">
  <color
    color="99efed">
    </color>
  </resource>

<!-- color used for disabled text -->
<resource
  id="clrdisable">
  <color
    color="808080">
    </color>
  </resource>

<!-- color used for summary text that should be a little dimmer than normal -->
<resource
  id="clrsummary">
  <color
    color="a0a0a0">
    </color>
  </resource>

<!-- color used for bright text -->
<resource
  id="clrbright">
  <color
    color="ffff88">
    </color>
  </resource>

<!-- color used for warning text -->
<resource
  id="clrwarning">
  <color
    color="ff0000">
    </color>
  </resource>

<!-- color used for prompt text inviting the user to change something -->
<resource
  id="clrprompt">
  <color
    color="ffff00">
    </color>
  </resource>

<!-- color used for the 'buy for free' checkbox on buy / sell panels -->
<resource
  id="clrchkfree">
  <color
    color="a8a800">
    </color>
  </resource>

<!-- color used for text on summary panels - a little dimmer than normal -->
<resource
  id="clrsummtxt">
  <color
    color="d0d0d0">
    </color>
  </resource>

<!-- color used for labels when choosing advancements -->
<resource
  id="clradvance">
  <color
    color="ffffff">
    </color>
  </resource>

<!-- color used for text on action buttons -->
<resource
  id="clraction">
  <color
    color="000088">
    </color>
  </resource>

<!-- colors used in edit controls -->
<resource
  id="clredittxt">
  <color
    color="d2d2d2">
    </color>
  </resource>
<resource
  id="clreditbck">
  <color
    color="000000">
    </color>
  </resource>

<!-- colors used in menu and chooser controls -->
<resource
  id="clrmenutxt">
  <color
    color="84c8f7">
    </color>
  </resource>
<resource
  id="clrmenuslt">
  <color
    color="1414f7">
    </color>
  </resource>
<resource
  id="clrmenubck">
  <color
    color="2a2c47">
    </color>
  </resource>
```

Numerous Places: Change the style to reference a named color instead of a literal color, switching from a "textcolor" attribute to a "textcolorid" attribute. The list of changes is given below.

- Line 282 - "clrtitle"
- Line 293 - "clrstatic"
- Line 303 - "clrnormal"
- Line 313 - "clrnormal"
- Line 323 - "clrnormal"
- Line 335 - "clrauto"
- Line 347 - "clrdisable"
- Line 359 - "clrbright"
- Line 371 - "clrwarning"
- Line 384 - "clrprompt"
- Line 394 - "clrnormal"
- Line 404 - "clrdisable"
- Line 414 - "clrprompt"
- Line 424 - "clrwarning"
- Line 434 - "clrnormal"
- Line 444 - "clrdisable"
- Line 454 - "clrwarning"
- Line 464 - "clrnormal"

- Line 474 - "clrdisable"
- Line 484 - "clrsummary"
- Line 494 - "clrnormal"
- Line 504 - "clrnormal"
- Line 514 - "clrsummtxt"
- Line 524 - "clrdisable"
- Line 534 - "clrsummary"
- Line 545 - "clrsummary"
- Line 556 - "clradvance"
- Line 568 - "clradvance"
- Line 582 - "clrnormal"
- Line 596 - "clredittxt"
- Line 597 - "clreditbck"
- Line 607 - "clredittxt"
- Line 608 - "clreditbck"
- Line 618 - "clredittxt"
- Line 619 - "clreditbck"
- Line 624 - "clrdisable"
- Line 635 - "clrnormal"
- Line 652 - "clrnormal"
- Line 669 - "clrnormal"
- Line 732 - "clrnormal"
- Line 805 - "clraction"
- Line 836 - "clraction"
- Line 867 - "clrnormal"
- Line 901 - "clraction"
- Line 937 - "clraction"
- Line 971 - "clraction"
- Line 981 - "clraction"
- Line 991 - "clraction"
- Line 1025 - "clraction"
- Line 1059 - "clraction"
- Line 1085 - "clrnormal"
- Line 1119 - "clrnormal"
- Line 1145 - "clrnormal"
- Line 1171 - "clrnormal"
- Line 1205 - "clrnormal"
- Line 1231 - "clrnormal"
- Line 1265 - "clrnormal"
- Line 1299 - "clrnormal"
- Line 1333 - "clrnormal"
- Line 1359 - "clrnormal"
- Line 1385 - "clrnormal"
- Line 1411 - "clrnormal"
- Line 1445 - "clrnormal"
- Line 1471 - "clraction"

- Line 1497 - "clraction"
- Line 1531 - "clraction"
- Line 1557 - "clraction"
- Line 1583 - "clraction"
- Line 1609 - "clraction"
- Line 1643 - "clraction"
- Line 1677 - "clraction"
- Line 1711 - "clraction"
- Line 1745 - "clraction"
- Line 1779 - "clraction"
- Line 1813 - "clraction"
- Line 1847 - "clraction"
- Line 1881 - "clraction"
- Line 1915 - "clraction"
- Line 1949 - "clraction"
- Line 1983 - "clraction"
- Line 2017 - "clrnormal"
- Line 2051 - "clrnormal"
- Line 2085 - "clrnormal"
- Line 2119 - "clrnormal"
- Line 2153 - "clrnormal"
- Line 2206 - "clrdisable"
- Line 2232 - "clrnormal"
- Line 2241 - "clrwarning"
- Line 2252 - "clrdisable"
- Line 2261 - "clrchkfree"
- Line 2270 - "clrnormal"
- Line 2311 - "clrnormal"
- Line 2358 - "clrmenutxt"
- Line 2359 - "clrmenubck"
- Line 2360 - "clrmenuslt"
- Line 2361 - "clrnormal"
- Line 2371 - "clrmenutxt"
- Line 2372 - "clrmenubck"
- Line 2373 - "clrmenuslt"
- Line 2374 - "clrnormal"
- Line 2375 - "clrwarning"
- Line 2386 - "clrmenutxt"
- Line 2387 - "clrmenubck"
- Line 2397 - "clrwarning"
- Line 2398 - "clrmenubck"

Line 470: Insert the new style definition shown below.

```
<!-- slightly smaller label that is left-aligned -->
<style
  id="lblSmlLeft">
  <style_label
    textcolorid="clrnormal"
    font="fntsmall"
    alignment="left">
    </style_label>
  </style>
```

Line 1467: Insert the new style definition shown below.

```
<!-- Style used on the master button -->
<style
  id="actMaster">
  <style_action
    textcolorid="clrnormal"
    font="fntactsml"
    up="actmastup" down="actmastdn" off="actmastup">
    </style_action>
  <resource
    id="actmastup"
    isbuiltin="yes">
    <bitmap
      bitmap="master_up.bmp"
      istransparent="yes">
      </bitmap>
    </resource>
  <resource
    id="actmastdn"
    isbuiltin="yes">
    <bitmap
      bitmap="master_down.bmp"
      istransparent="yes">
      </bitmap>
    </resource>
  </style>
```

Line 2380: Insert the the new style definitions shown below.

```
<!-- small menu portal -->
<style
  id="menuSmall"
  border="sunken">
  <style_menu
    textcolorid="clrmenutxt"
    backcolorid="clrmenubck"
    selecttextid="clrmenuslt"
    selectbackid="clrnormal"
    font="fntmenusm"
    droplist="menuarrsm"
    droplistoff="menuoffsm">
    </style_menu>
  <resource
    id="menuarrsm"
    isbuiltin="yes">
    <bitmap
      bitmap="menu_small_arrow.bmp">
      </bitmap>
    </resource>
  <resource
    id="menuoffsm"
    isbuiltin="yes">
    <bitmap
      bitmap="menu_small_arrow_off.bmp">
      </bitmap>
    </resource>
  </style>

<!-- small menu portal with coloring to indicate contents are in error -->
<style
  id="menuErrSm"
  border="sunken">
  <style_menu
    textcolorid="clrmenutxt"
    backcolorid="clrmenubck"
    selecttextid="clrmenuslt"
    selectbackid="clrnormal"
    activetextid="clrwarning"
    font="fntmenusm"
    droplist="menuarrsm"
    droplistoff="menuoffsm">
    </style_menu>
  </style>
```

File: "form_advance.dat"

Line 28: Change the field reference from "name" to "thingname".

Line 123: Change the field reference from "name" to "thingname".

Line 222: Replace the code for shrinking the size to the new code shown below.

```
perform portal[name].sizetofit[36]
perform portal[name].centervert
```

Line 238: Replace the "label" element with the new behavior shown below.

```
<label>
  <labeltext><![CDATA[
    ~use any domain term specified, else default to "domain"
    @text = parent.tagnames[DomainTerm.?]
    if (empty(@text) <> 0) then
      @text = "Domain"
      endif
    @text &= ":"
    ]]></labeltext>
  </label>
```

Line 271: Insert the new template definition shown below.

```
<template
  id="advNotate"
  name="Notation Specification"
  compset="AdvDetails">

  <portal
    id="lblnotate"
    style="lblStatic">
    <label
      text="Notation:">
      </label>
    </portal>

  <portal
    id="notation"
    style="editNormal">
    <edit
      field="advUser"
      maxlength="50">
      </edit>
    </portal>

  <position><![CDATA[
    ~set up our width and height
    height = portal[notation].height
    if (issizing <> 0) then
      done
      endif

    ~center everything vertically
    perform portal[lblnotate].centervert
    perform portal[notation].centervert

    ~position the label on the left with the edit portal next to it
    portal[lblnotate].left = 0
    perform portal[notation].alignrel[ltor,lblnotate,7]
    portal[notation].width = width - portal[notation].left
    ]]></position>
  </template>
```

Line 309: Replace the script code for setting the line height to the code shown below.

```
portal[notes].lineheight = 4
```

Line 320: Insert the template reference shown below.

```
<templateref template="advNotate" thing="advDetails" taborder="20"/>
```

Line 345: Insert the code below to position the notation template

```
~position the notation template in the same place
template[advNotate].left = portal[advNew].left
template[advNotate].top = portal[advNew].top
template[advNotate].width = portal[advNew].width
```

Line 347: Insert the code below to set the notation template to non-visible.

```
template[advNotate].visible = 0
```

Line 356: Insert the code below to show the notation template when appropriate.

```
elseif (container.parent.tagis[Advance.Notation] <> 0) then
  template[advNotate].visible = 1
```

File: "form_dashboard.dat"
Line 121: Replace the Label script code with the new code shown below.

```
@text = "{size 30}PP: {size 36}" & hero.child[trkPower].field[trkUser].text
```

Line 189: Added the "isbuiltin" attribute to the element with a value of "yes".

File: "form_static.dat"
Line 59: Insert the new portal definition below.

```
<portal
  id="master"
  style="actMaster"
  tiptext="Click this button to activate this ally's Master.">
  <action
    action="master">
    </action>
  </portal>
```

Line 76 and 87: Put the contents of the Position script into a "<![CDATA[...]]>" block.

Line 77: Insert the code below at the start of the Position script.

```
~only show the master button if the actor is a minion
portal[master].visible = hero.isminion
if (portal[master].visible <> 0) then
  perform portal[label].alignrel[ltor,master,8]
  endif
```

File: "form_taccon.dat"
Line 179: Insert the "isbuiltin" attribute with a value of "yes".

Line 189: Insert the "isbuiltin" attribute with a value of "yes".

Line 199: Insert the "isbuiltin" attribute with a value of "yes".

Line 203: Insert the new portal definition shown below.

```
<portal
  id="dead"
  style="imgNormal"
  tiptext="This character is dead or otherwise out of combat.">
  <image_literal
    image="tactical_dead.bmp"
    isbuiltin="yes"
    istransparent="yes">
    </image_literal>
  </portal>
```

Line 216: Change the style assigned to the portal to "lblSmlLeft".

Line 225: Eliminate the left-alignment logic from the Label script as shown below.

```
~squeeze inter-line spacing a bit
@text = "{leading -2}" & @text
```

Line 254: Replace the "damage" and "status" portals with the new portal definitions shown below.

```
<portal
  id="status1"
  style="lblSmlLeft">
  <label
    ismultiline="yes">
    <labeltext><![CDATA[
      ~start with the power points status
      @text = "{size 30}PP: {size 36}" & #traituser[trPowerPts] & " / " & #trait[trPowerPts]

      ~add the defense rating
      @text &= "{horz 12}{size 30}Def: {size 36}" & #trait[trDefense]
      ]]></labeltext>
    </label>
  </portal>

<portal
  id="status2"
  style="lblSmlLeft">
  <label
    ismultiline="yes">
    <labeltext><![CDATA[
      @text = "{size 30}HP: {size 36}" & field[acHPSumm].text
      ]]></labeltext>
    </label>
  </portal>
```

Line 451: Renamed the portal from "column1" to "traits".

Line 461: Change the "foreach" statement to reference the "DashTacCon.Traits" tag expression.

Line 517: Append "sortas _NameSeq_" to the end of the "foreach" statement.

Line 544: Replace the "foreach" statement with the new statement shown below.

```
foreach pick in hero where "(Adjustment.? | Helper.Activated) & !InPlay.Permanent" sortas
_NameSeq_
  if (ismore <> 0) then
    @text &= "; "
    endif
  if (eachpick.tagis[component.Adjustment] <> 0) then
    @text &= eachpick.field[adjShort].text
  elseif (eachpick.tagis[component.shortname] <> 0) then
    @text &= eachpick.field[shortname].text
  else
    @text &= eachpick.field[name].text
    endif
  ismore = 1
  nexteach
```

Line 669-681: Replace the script code that controls visibility and positioning with the new logic shown below.

```
~position the "dead" indicator in the same location
portal[dead].left = leftedge
perform portal[dead].centervert

~hide all of the indicators and we'll pick one to show below (or none)
portal[dead].visible = 0
portal[acted].visible = 0
portal[noncombat].visible = 0
portal[never].visible = 0

~if we're in combat, handle things appropriately
if (state.iscombat <> 0) then

  ~determine which of the above four indicators is actually visible
  if (hero.tagis[Hero.Dead] <> 0) then
    portal[dead].visible = 1
  else
    portal[never].visible = hero.tagis[combat.never]
    portal[acted].visible = hero.tagis[combat.acted]
    portal[noncombat].visible = hero.tagis[combat.noncombat]
    endif

  ~adjust our left edge rightward past the indicators
  leftedge += portal[never].width + 4
  endif
```

Line 759-778: Replace the script code that references the renamed "damage" and "status" portals with the new code shown below.

```
~position the second status portal at the bottom of the region
perform portal[status2].alignedge[bottom,-margin - 2]
portal[status2].left = portal[name].left + 3

~position the first status portal in the region above the second status portal
perform portal[status1].alignrel[btot,status2,-1]
portal[status1].left = portal[status2].left

~if the status now overlaps the name, shift the status portals downward a
~little bit to make additional space
if (portal[status2].top < portal[name].bottom) then
  var adjust as number
  adjust = portal[name].bottom - portal[status2].top
  portal[status2].top += adjust
  adjust -= 1
  if (adjust > 1) then
    adjust -= 1
    endif
  portal[status1].top += adjust
  endif
```

Line 851: Replace the code manipulating the "column1" portal with the following.

```
~position the column of reminder traits
portal[traits].top = margin + 1
portal[traits].left = leftedge
portal[traits].lineheight = 3
```

Line 860: Replace the reference to the "column1" portal with the "traits" portal.

Line 868: Insert the new code below to size the summary appropriately.

```
perform portal[summary].sizetofit[28]
```

Line 870-871: Replace the references to the "column1" portal with the "traits" portal.

File: "procedures.dat"

Line 780: Replace the "foreach" statement with the new logic shown below.

```
foreach pick in hero where "(Adjustment.? | Helper.Activated) & !InPlay.Permanent"
  final &= "{br}"
  if (eachpick.tagis[Adjustment.?] <> 0) then
    final &= eachpick.field[adjName].text
  else
    final &= eachpick.field[name].text
    endif
  nexteach
```

File: "sheet_standard1.dat"

Line 485: Replace the "dots" portal with the new definition below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Line 627: Insert the new logic below into the Position script.

```
~limit our portal height to a single line of output
```

```
portal[details].lineheight = 1
```

Line 661: Replace the "dots" portal with the new definition below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Line 670: Replace the entire Position script with the new logic shown below.

```
~our height is the height of the tallest portal
height = portal[name].height
if (issizing <> 0) then
  done
  endif

~position the value at the right edge
perform portal[value].alignedge[right,0]

~size the name to fit the available space
portal[name].width = portal[value].left - 10
perform portal[name].sizetofit[40]
perform portal[name].autoheight

~the dots should span the region between the name and the value
perform portal[dots].alignrel[ltor,name,5]
portal[dots].width = portal[value].left - 5 - portal[dots].left

~center all portals vertically
perform portal[name].centervert
perform portal[value].centervert
perform portal[dots].centervert
```

Line 711: Replace the entire Label script with the new logic shown below.

```
if (stackable = 0) then
  @text = ""
elseif (field[stackQty].value = 1) then
  @text = ""
else
  @text = field[stackQty].text & "x"
  endif
```

Line 731: Replace the entire Position script with the new logic shown below.

```
~our height is the height of the tallest portal
height = portal[name].height
if (issizing <> 0) then
  done
  endif

~assign a fixed width to the value and position the name to the right
portal[value].width = 100
perform portal[name].alignrel[ltor,value,20]

~size the name to fit the available space
portal[name].width = width - portal[name].left
perform portal[name].sizetofit[36]
perform portal[name].autoheight

~center all portals vertically
perform portal[value].centervert
perform portal[name].centervert
```

Lines 811-826: Replace the block of script code with the new logic shown below.

```
~align everything horizontally
perform portal[badstr].alignrel[ltor,equipped,5]
perform portal[name].alignrel[ltor,badstr,5]
```

```
~size the name to fit the available space
portal[name].width = width - portal[name].left
perform portal[name].sizetofit[36]
perform portal[name].autoheight

~center all portals vertically
perform portal[badstr].centervert
perform portal[equipped].centervert
perform portal[name].centervert

~shift the "equipped" bitmap downward a little bit; this is because it is a
~lone bitmap drawn via encoded text, and bitmaps are never drawn within the
~descender portion of the text, which causes it to appear higher than we want it
portal[equipped].top += 4
```

Line 867: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="outNameMed">
  <output_label
    field="shortname">
    </output_label>
  </portal>
```

Line 912: Insert the new portal shown below.

```
<portal
  id="dots"
  style="outDots">
  <output_dots>
    </output_dots>
  </portal>
```

Lines 946-989: Replace the entire Position script with the new logic shown below.

```
~our height is based on the tallest portal within
height = portal[name].height
if (issizing <> 0) then
  done
  endif

~if the weapon satisfies the minimum strength requirement, hide the bitmap
if (tagis[Helper.BadStrReq] = 0) then
  portal[badstr].visible = 0
  endif

~center all portals vertically
perform portal[badstr].centervert
perform portal[name].centervert
perform portal[attack].centervert
perform portal[damage].centervert
perform portal[dots].centervert

~position the range with the same baseline as the rest of the text; since it
~uses a smaller font, it will have a smaller height, so centering it will have
~it appear to float up relative to the other text
perform portal[range].alignrel[btob,name,0]

~establish suitable fixed widths for the various columns of data
portal[damage].width = 120
portal[attack].width = 70
portal[range].width = 260

~position everything horizontally, leaving a margin on both sides appropriately
portal[badstr].left = 5
perform portal[damage].alignedge[right,-5]
perform portal[name].alignrel[ltor,badstr,5]
perform portal[attack].alignrel[rtol,damage,-10]
perform portal[range].alignrel[rtol,attack,-10]

~if this is a ranged weapon, limit the name to the space up to the range details;
~otherwise, let the name extend over to the attack value
var limit as number
if (tagis[component.WeapRange] <> 0) then
  limit = portal[range].left
```

```
else
  limit = portal[attack].left
  endif

~limit the name to the extent determined above
if (portal[name].right > limit - 5) then
  portal[name].width = limit - portal[name].left - 5
  endif

~size the name to fit the available space
perform portal[name].sizetofit[36]
perform portal[name].autoheight
perform portal[name].centervert

~extend the dots from the right of the name across to the value on the right
if (portal[name].right > limit - 10) then
  portal[dots].visible = 0
else
  perform portal[dots].alignrel[ltor,name,5]
  portal[dots].width = limit - 5 - portal[dots].left
  endif
```

Line 1039: Replace the entire Position script with the new logic shown below.

```
~our height is the vertical extent of our portals
height = portal[name].height
if (issizing <> 0) then
  done
  endif

~size the name to fit the available space
portal[name].width = width
perform portal[name].sizetofit[36]
perform portal[name].autoheight
perform portal[name].centervert
```

Line 1173: Replace the code for calculating the validation report height with the new logic below.

```
perform portal[validate].autoheight
if (portal[validate].height > portal[validate].fontheight * maxlines) then
  portal[validate].lineheight = maxlines
  endif
```

Numerous Places: Changes references to the "global" target reference over to "scenevalue". The list of locations is given below.

- Line 1208
- Line 1229
- Line 1315
- Line 1349
- Line 1442
- Line 1446
- Line 1464
- Line 1479
- Line 1480
- Line 1490

File: "sheet_standard2.dat"
Line 70: Changes reference to the "global" target reference over to "scenevalue".

File: "summ_armory.dat"
Line 76: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="lblSummary">
  <label>
    <labeltext><![CDATA[
      if (field[grIsEquip].value <> 0) then
```

```
        @text = "{b}{i}"
        endif
      @text &= field[name].text
      ]]></labeltext>
    </label>
  <mouseinfo/>
  </portal>
```

Line 118: Replace the "name" portal with the new element shown below.

```
<portal
  id="name"
  style="lblSummary">
  <label>
    <labeltext><![CDATA[
      if (field[grIsEquip].value <> 0) then
        @text = "{b}{i}"
        endif
      @text &= field[name].text
      ]]></labeltext>
    </label>
  <mouseinfo/>
  </portal>
```

File: "tab_armory.dat"
Line 86: Change the component referenced from "WeapRange" to "Gear".

Line 265: Insert the "isbuiltin" attribute with a value of "yes".

Line 277: Insert the "isbuiltin" attribute with a value of "yes".

Line 319: Insert the "isbuiltin" attribute with a value of "yes".

Line 519: Insert the "isbuiltin" attribute with a value of "yes".

Line 531: Insert the "isbuiltin" attribute with a value of "yes".

File: "tab_gear.dat"
Line 122: Change the style from "lblNormal" to "lblLeft".

Line 143: Insert the "isbuiltin" attribute with a value of "yes".

Line 155: Insert the "isbuiltin" attribute with a value of "yes".

Line 198: Replace the entire Position script with the new logic shown below.

```
~set up our height based on our tallest portal
height = portal[info].height

~if this is a "sizing" calculation, we're done
if (issizing <> 0) then
  done
  endif

~determine whether the container and heldby indicators should be visible
portal[container].visible = tagis[thing.holder?]
portal[heldby].visible = isgearheld

~center the portals vertically
perform portal[info].centervert
perform portal[name].centervert
perform portal[username].centervert
perform portal[gearmanage].centervert
perform portal[delete].centervert
perform portal[container].centervert
perform portal[heldby].centervert

~position the delete portal on the far right
perform portal[delete].alignedge[right,0]

~position the info portal to the left of the delete button
perform portal[info].alignrel[rtol,delete,-8]

~position the gear portal to the left of the info button
perform portal[gearmanage].alignrel[rtol,info,-8]

~calculate the space to reserve for the various indicators
var reserve as number
if (portal[heldby].visible <> 0) then
```

```
  reserve += portal[heldby].width + 2
  endif
if (portal[container].visible <> 0) then
  reserve += portal[container].width + 2
  endif
if (portal[heldby].visible + portal[container].visible <> 0) then
  reserve += 3
  endif

~position the name on the left and let it use all available space
var limit as number
limit = portal[gearmanage].left - 8 - reserve
portal[name].left = 0
portal[name].width = minimum(portal[name].width,limit)

~if this is a "custom" gear pick, show an edit portal instead of the name
var nextleft as number
if (tagis[Equipment.CustomGear] <> 0) then
  portal[name].visible = 0
  portal[username].left = portal[name].left
  portal[username].width = minimum(200,limit)
  nextleft = portal[username].right
else
  portal[username].visible = 0
  nextleft = portal[name].right
  endif
nextleft += 5

~show the 'container' icon to the right of the name (if visible)
if (portal[container].visible <> 0) then
  portal[container].left = nextleft
  nextleft = portal[container].right + 2
  endif

~show the 'held by' icon to the right of the container icon (if visible)
if (portal[heldby].visible <> 0) then
  portal[heldby].left = nextleft
  endif

~if the gear can't be deleted (i.e. it's been auto-added instead of user-added,
~set the style to indicate that behavior to the user
if (candelete = 0) then
  perform portal[name].setstyle[lblAuto]
  endif
```

File: "tab_journal.dat"

Line 98: Replace the line of code that references the usage pool to the new line shown below.

```
@text &= "{horz 40} Total XP: " & #resmax[resXP]
```

File: "thing_abilities.dat"

Line 23: Insert the following material within the "abSample" ability.

```
<!-- If checkbox selection is needed, make sure the compset includes "UserSelect"
      component and define this field appropriately.
<fieldval field="usrChkText" value="Menu1"/>
-->

<!-- If thing-based menu selection is needed, make sure the compset includes
      "UserSelect" component and define these fields and tags as appropriate.
<fieldval field="usrLabel1" value="Menu1"/>
<fieldval field="usrCandid1" value="component.Attribute"/>
<fieldval field="usrLabel2" value="Menu2"/>
<fieldval field="usrCandid2" value="component.Skill"/>
<tag group="ChooseSrc1" tag="Hero"/>
<tag group="ChooseSrc2" tag="Thing"/>
-->

<!-- If array-based menu selection is needed, make sure the compset includes
      "UserSelect" component and define these fields as appropriate.
<fieldval field="usrLabelAr" value="Menu1"/>
<arrayval field="usrArray" index="0" value="Choice #1"/>
<arrayval field="usrArray" index="1" value="Choice #2"/>
<arrayval field="usrArray" index="2" value="Choice #3"/>
-->
```

File: "thing_armory.dat"
Line 18: Insert the following thing definition.

```
<!-- Natural armor is automatically equipped -->
<thing
  id="armNatural"
  name="Natural Armor"
  compset="Armor"
  description="Description goes here"
  isunique="yes"
  holdable="no">
  <fieldval field="defDefense" value="1"/>
  <tag group="Equipment" tag="Natural"/>
  <tag group="Equipment" tag="AutoEquip"/>
  </thing>
```

File: "thing_attributes.dat"
Line 18: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sam"/>
```

Line 31: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Str"/>
```

File: "thing_skills.dat"
Line 18: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sam"/>
```

Line 37: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Mel"/>
```

Line 50: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Sht"/>
```

File: "thing_traits.dat"
Line 38: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Hlth"/>
```

Line 60: Change the "Column1" tag reference to "Traits".

Line 77: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Def"/>
```

Line 96: Insert the following field value assignment.

```
<fieldval field="trtAbbrev" value="Powr"/>
```

Line 117: Change the "Column1" tag reference to "Traits".

Line 147: Change the "Column1" tag reference to "Traits".

File: "visual.dat"
Line 130: Insert the following code at the end of the Position script.

```
~center the name if requested
if (tagis[SimpleItem.CenterName] <> 0) then
  perform portal[name].centerhorz
  endif
```

```
~if this is an auto-added pick, change its font to indicate that fact
if (ispick + !candelete >= 2) then
  perform portal[name].setstyle[lblAuto]
  endif
```

Line 213: Insert the new template definition below.

```
<!-- UserSelect template
      Similar to SimpleItem, except that this template is only suitable for
      showing picks and the items can employ various mechanisms for customizing
      the contents of the pick in some way. This template can be used or readily
      adapted when you integrate the "UserSelect" component into a component set
      and want to let the user customize the pick contents. For more details,
      please refer to the "UserSelect" component.
-->
<template
  id="UserSelect"
  name="User Selection"
  compset="UserSelect"
  marginhorz="3"
  marginvert="2">

  <portal
    id="name"
    style="lblNormal"
    showinvalid="yes">
    <label
      field="thingname">
      </label>
    </portal>

  <portal
    id="lblmenu1"
    style="lblSecond">
    <label
      field="usrLabel1">
      </label>
    </portal>

  <portal
    id="lblmenu2"
    style="lblSecond">
    <label
      field="usrLabel2">
      </label>
    </portal>

  <portal
    id="menu1"
    style="menuNormal">
    <menu_things
      field="usrChosen1"
      component="none"
      maxvisible="10"
      usepicksfield="usrSource1"
      candidatefield="usrCandid1">
      </menu_things>
    </portal>

  <portal
    id="menu2"
    style="menuNormal">
    <menu_things
      field="usrChosen2"
      component="none"
      maxvisible="10"
      usepicksfield="usrSource2"
      candidatefield="usrCandid2">
      </menu_things>
    </portal>

  <portal
    id="lblmenuar"
    style="lblSecond">
    <label
      field="usrLabelAr">
      </label>
    </portal>
```

```
<portal
  id="menuarray"
  style="menuNormal">
  <menu_array
    field="usrSelect"
    array="usrArray"
    maxvisible="10">
    </menu_array>
  </portal>

<portal
  id="checkbox"
  style="chkNormal">
  <checkbox
    field="usrIsCheck"
    dynamicfield="usrChkText">
    </checkbox>
  </portal>

<portal
  id="info"
  style="actInfo">
  <action
    action="info">
    </action>
  <mouseinfo/>
  </portal>

<portal
  id="delete"
  style="actDelete"
  tiptext="Click to delete this item">
  <action
    action="delete">
    </action>
  </portal>

<position><![CDATA[
  ~set up our height based on our tallest portal
  height = portal[info].height

  ~if this is a "sizing" calculation, we're done
  if (issizing <> 0) then
    done
    endif

  ~position our tallest portal at the top
  portal[info].top = 0

  ~center the other portals vertically
  perform portal[name].centervert
  perform portal[delete].centervert
  perform portal[lblmenu1].centervert
  perform portal[menu1].centervert
  perform portal[lblmenu2].centervert
  perform portal[menu2].centervert
  perform portal[lblmenuar].centervert
  perform portal[menuarray].centervert
  perform portal[checkbox].centervert

  ~determine whether our portals are visible; we only show them if requested
  ~Note: Remember that a non-empty tagexpr field indicates menu selection is used.
  if (field[usrCandid1].isempty <> 0) then
    portal[lblmenu1].visible = 0
    portal[menu1].visible = 0
  elseif (field[usrLabel1].isempty <> 0) then
    portal[lblmenu1].visible = 0
    endif
  if (field[usrCandid2].isempty <> 0) then
    portal[lblmenu2].visible = 0
    portal[menu2].visible = 0
  elseif (field[usrLabel2].isempty <> 0) then
    portal[lblmenu2].visible = 0
    endif
  if (empty(field[usrArray].arraytext[0]) <> 0) then
    portal[lblmenuar].visible = 0
    portal[menuarray].visible = 0
  elseif (field[usrLabelAr].isempty <> 0) then
    portal[lblmenuar].visible = 0
    endif
  if (field[usrChkText].isempty <> 0) then
```

```
    portal[checkbox].visible = 0
    endif

  ~position the delete portal on the far right and the info portal next to it
  perform portal[delete].alignedge[right,0]
  perform portal[info].alignrel[rtol,delete,-8]

  ~determine our effective right edge, allowing for the buttons above
  var edge as number
  edge = portal[info].left - 10

  ~setup the default portal width and gap to be used between and around portals
  var defwidth as number
  var gap as number
  defwidth = 100
  gap = 10

  ~determine the minimum amount of space we need to reserve for our portals
  var reserve as number
  if (portal[checkbox].visible <> 0) then
    reserve = defwidth
  elseif (portal[menuarray].visible <> 0) then
    reserve = portal[lblmenuar].width * portal[lblmenuar].visible
    reserve += defwidth + gap
  elseif (portal[menu1].visible <> 0) then
    reserve = portal[lblmenu1].width * portal[lblmenu1].visible
    reserve += defwidth + gap
    reserve += portal[lblmenu2].width * portal[menu2].visible
    reserve += (defwidth + gap) * portal[menu2].visible
    endif

  ~position the name on the left, reserving our minimum space for any portals
  var x as number
  portal[name].left = 0
  portal[name].width = minimum(portal[name].width,edge - portal[name].left - reserve)
  x = portal[name].right + gap

  ~setup the maximum width for our some portals, regardless of space available
  var maxwidth as number
  maxwidth = 150

  ~if we have a checkbox, size and position it appropriately
  if (portal[checkbox].visible <> 0) then
    portal[checkbox].left = x

  ~if we have an array-based menu, size and position it appropriately
  elseif (portal[menuarray].visible <> 0) then
    if (portal[lblmenuar].visible <> 0) then
      portal[lblmenuar].left = x
      x = portal[lblmenuar].right + 4
      endif
    portal[menuarray].left = x
    portal[menuarray].width = maxwidth

  ~if we have one thing-based menu, size and position it appropriately
  elseif (portal[menu1].visible + portal[menu2].visible = 1) then
    if (portal[lblmenu1].visible <> 0) then
      portal[lblmenu1].left = x
      x = portal[lblmenu1].right + 4
      endif
    portal[menu1].left = x
    portal[menu1].width = minimum(edge - portal[menu1].left,maxwidth)

  ~if we have two thing-based menus, size and position them appropriately
  elseif (portal[menu1].visible <> 0) then
    if (portal[lblmenu1].visible <> 0) then
      portal[lblmenu1].left = x
      x = portal[lblmenu1].right + 4
      endif
    portal[menu1].left = x
    var extra as number
    extra = (portal[lblmenu2].width + 4) * portal[lblmenu2].visible
    var actual as number
    actual = (edge - portal[menu1].left - extra - gap) / 2
    portal[menu1].width = minimum(actual,maxwidth)
    portal[menu2].width = portal[menu1].width
    x = portal[menu1].right + gap
    if (portal[lblmenu2].visible <> 0) then
      portal[lblmenu2].left = x
      x = portal[lblmenu2].right + 4
      endif
    portal[menu2].left = x
    endif

  ~if a menu is visible, make sure it has a selection
  if (portal[menu1].visible <> 0) then
    if (field[usrChosen1].ischosen = 0) then
      perform portal[menu1].setstyle[menuError]
      endif
    endif
  if (portal[menu2].visible <> 0) then
    if (field[usrChosen2].ischosen = 0) then
      perform portal[menu2].setstyle[menuError]
      endif
    endif
  if (portal[menuarray].visible <> 0) then
    if (field[usrSelect].isempty <> 0) then
      perform portal[menuarray].setstyle[menuError]
      endif
    endif
  ]]></position>

</template>
```

Category: Kit Reference

Skeleton File Changes V3.2

Context: HL Kit ... Kit Reference ... Skeleton Data File Revision History

Contents

File: "tags.1st"
Line 39: Added definition of the "Helper.NoMinimum" tag for use with Resources.

File: "equipment.str"
Line 68: Corrected the calculation of the "lot cost" for a piece of gear.

Line 767: When creating ammunition and we have a transaction pick, the initial user quantity must be setup based on the actual quantity purchased.

File: "traits.str"
Line 214: Added the "Activated" identity tag group for shared use with adjustments.

Line 240: If an ability is activated, the corresponding "Activated" identity tag is now forwarded to the actor.

File: "miscellaneous.str"
Line 182: Added new Eval Rule script that reports a validation warning for any resource that is not fully spent, except if that resource is assigned the "Helper.NoMinimum" tag.

File: "thing_miscellaneous.dat"
Line 81: Assigned "Helper.NoMinimum" tag to disable reporting a validation warning if XP are not spent.

File: "form_static.dat"
Line 81: Eliminated the "maxlength" attribute that now results in a compilation error.

File: "form_config.dat"
Line 92: Eliminated the "maxlength" attribute that now results in a compilation error.

File: "form_manipulate.dat"
Line 49: Eliminated the "maxlength" attribute that now results in a compilation error.

File: "sheet_standard2.dat"
Line 85: The right-side column of output must be properly positioned on the right.

File: "editor.dat"
Line 17: Added "prefix" attribute for use by the Editor.

Line 42: Added "prefix" attribute for use by the Editor.

Line 72: Added "prefix" attribute for use by the Editor.

Line 119: Added "prefix" attribute for use by the Editor.

Line 129: Added "prefix" attribute for use by the Editor.

Line 149: Added "prefix" attribute for use by the Editor.

Line 194: Added "prefix" attribute for use by the Editor.

Line 254: Added "prefix" attribute for use by the Editor.

Line 279: Added "prefix" attribute for use by the Editor.

Line 156: Added "Ammunition" edit thing to allow users to enter ammunition via the Editor.

Line 137: Added "Lot Cost" and "Weight" fields for use by the Editor.

Line 207: Added "Weight" field for use by the Editor.

Line 272: Added "Weight" field for use by the Editor.

Line 317: Added "Weight" field for use by the Editor.

Line 347: Added "Weight" field for use by the Editor.

File: "system_resources.aug"
Line 268: Added definition of the "tabwarning" color resource to allow authors to easily override the resource if they wish.

Line 626: Added definition of the "edtbackov" color resource to enable overrides.

Line 1131: Added definition of the "progtext" color resource to enable overrides.

Line 1315: Added definition of the "cnflictext" color resource to enable overrides.

Line 1315: Added definition of the "buygameup" and "buygamedn" bitmap resources to enable overrides.

Category: Kit Reference

XML Character Encoding Set

Context: HL Kit ... Kit Reference

HL assumes all XML documents utilize the "ISO-8859-1" character set (also called Latin-1), with a number of exceptions specific to the Windows platform. The list of exceptions is detailed in the table below.

128	undefined
129	undefined
130	‚
131	ƒ
132	„
133	…
134	†
135	‡
136	ˆ
137	‰
138	Š
139	‹
140	Œ
141	undefined
142	undefined
143	undefined
144	undefined
145	‘
146	’
147	“
148	”
149	•
150	–
151	—
152	˜
153	™
154	š
155	›
156	œ
157	undefined
158	undefined
159	Ÿ

The identity element at the top of all XML files should specify an encoding of "ISO-8859-1" for completeness. If no encoding is given, ISO-8859-1 is assumed. An example is given below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

NOTE! There is an unofficial XML encoding named "Windows-1252" that properly reflects the Windows ANSI character set and is often used. However, various XML parsers do not recognize this encoding set due to its unofficial nature. In the interest of maximum compatibility, the modified Latin-1 set is used instead.

Category: Kit Reference

XML Attributes in Data Files

Context: HL Kit … Kit Reference

All of the information contained within data files is managed through XML elements, attributes, and PCDATA. Each XML element typically has a number of attributes which dictate the characteristics of that element. Some attributes will be required and some will be optional. When an attribute is optional, it will be clearly designated as such, along with the default value. Unless specified as optional, all attributes are required.

The XML specification allows attributes to represent virtually anything with a text value. However, the Kit differentiates between various types of attributes, each of which has various rules associated with the values it can be assigned. Within the Kit Reference, each attribute's type will be specified in its description. The following types of attributes are utilized, with the indicated constraints being imposed.

Text	Text attributes must always consist of standard ASCII text characters. Unless specified otherwise in the description of the attribute, text attributes have no additional constraints other than those dictated by the use of the attribute's contents. For example, a text attribute that specifies a file name needs to comply with the naming rules for files, else attempts to use the file name will fail. Any special constraints are indicated within the attribute description.
Integer	Integer attributes must specify an integer value. In general, integer attributes will represent positive numbers, although it is valid to specify a negative value when appropriate.
Float	Float attributes must specify a floating point value (e.g. 3.14159). In general, only positive values are used, although negative values are sometimes allowed. The attribute description will specify any restrictions.
Boolean	Boolean attributes must specify a value of either "yes" or "no". The name of the Boolean attribute will typically give a clear indication of the meaning of "yes" or "no". For example, tag groups have a "visible" attribute, where the meanings of "yes" and "no" are clear.
Id	Ids represent the unique id of an object. The rules for Unique Ids are outlined separately.
Set	Set-based attributes must specify one of an explicit set of values. Boolean attributes are an example of a set-based attribute, wherein the value must be specified as one of the two valid choices: "yes" and "no". Set-based attributes will specify the valid values and their meanings.

NOTE! Many attributes of type "text" will have an appropriate maximum length specified in their description. It is perfectly valid to specify a value that exceeds the indicated maximum within an XML file, since XML imposes no constraints. However, if you exceed the designated maximum length for a text attribute, HL will either report an error or simply truncate the text to the maximum length allowed (depending on the context). This can potentially yield unintended results, so be careful when specifying text attributes. Some text attributes have no maximum length, in which case HL will utilize whatever you specify for the attribute.

Category: Kit Reference

Specifying PCDATA in Data Files

Numerous elements throughout the various XML files utilize the PCDATA block of the element to hold important information. PCDATA blocks, in conjunction with the "<[CDATA[" section, make it possible to easily enter lengthy text that does not need to worry about the various XML restrictions on certain characters used in the text.

To simplify the documentation, the use of PCDATA within an element will not be handled separately. Instead, any element that uses PCDATA will simply have an attribute listed with the name "PCDATA". The description for this attribute will apply to the use of the PCDATA block within the element.

One characteristic of PCDATA is that it can optionally preserve or collapse all use of whitespace characters within the block. Whitespace characters include spaces, tabs, and newlines. When whitespace is collapsed, text that runs across multiple lines is merged together into a single flow, which is ideal when you're writing paragraphs of information, such as release notes. However, there are many times when it is critical that whitespace be preserved, such as with scripts that require each statement to be on a separate line.

When a PCDATA block is used as a script, the attribute type will be designated as "Script" to distinguish it from the normal "Text" type. By default, whitespace will be preserved for scripts and collapsed for all other situations. Any exceptions to this behavior will be clearly specified within the description for the PCDATA attribute.

Optional Attributes in Data Files

Context: HL Kit … Kit Reference

Information that is defined through XML attributes and/or PCDATA blocks is always required by default. The exception is an entry that is specifically designated as optional. If an entry is optional, it will be clearly designated as such within its description, with the text "(Optional)" appearing before the type is given. An optional attribute will always specify the default value that is used if the attribute is left blank. Optional PCDATA blocks are always treated as empty by default.

Category: Kit Reference

Leveraging Tags Via Tag Expressions

Context: HL Kit ... Kit Reference

Contents

General Overview

The Kit utilizes tags as a fundamental building block upon which a substantial number of mechanisms are based. While scripts and field values are critical components as well, you'll often rely on tags to determine when and how to process those scripts and fields. This section provides in-depth details on how to leverage tags in your design and how tags are utilized through tag expressions.

In the topics below, we focus on the various ways in which tags can be utilized. The syntax details are quite similar for each, since all are built upon the same fundamental elements. There are four basic layers of use that progressively build upon each other.

Simply click on the topics below to learn more about it.

IMPORTANT! This section assumes you are already familiar with the basics of tags and tag expressions. If you have not already done so, please review the section Data Manipulation Basics before proceeding with this section.

Tag Templates

Tag templates provide a simple mechanism to determine if an entity contains a specific tag. Templates also provide wildcard comparisons.

Tag Terms

Tag terms build upon tag templates to determine if entities comply with a specific criteria, whether it be the presence of a tag or an arithmetic comparison against a tag.

Tag Expressions

Tag expressions utilize Boolean logic to combine tag terms into complex expressions.

Arithmetic Expressions

Arithmetic expressions enable the creation of calculations based upon the tags assigned to objects in the portfolio.

Category: Kit Reference

Tag Templates

Context: HL Kit … Basic Concepts and Terminology … Tags and Tag Expressions

Contents

Overview

All things are assigned an assortment of tags that reflect the various characteristics of those things. Tags can be utilized by the data file author to identify things that must adhere to rules or should be subject to specific types of processing. You can assign tags to picks and containers dynamically during evaluation processing to change their nature and behavior. Tags can even be utilized by the end-user to filter things in arbitrary combinations. So now the question is how to identify the things that have the specific tags you are seeking. The basic mechanism is called a tag template.

Basic Usage and Syntax

Tag templates represent a kind of "query" that compares an object against a specific tag, or possibly a group of similar tags using a wildcard. Either the object contains one or more tags that match the template or not. In general, a tag template returns the number of tags within the object that match the template. How that value is used depends on the context in which it is used.

Tag templates identify a specific tag group and a comparison string that can match zero or more tags within that group. The syntax for a tag template is "group.compare", where "group" is the unique id of a specific tag group and "compare" is a comparison string to match against individual tags within the group. The two values are always separated by a period. The comparison string always compares against the unique id assigned to tags.

Note that tag templates always utilize the unique ids of the tag and the group. This makes tag usage completely independent of that actual names used for tags and/or tag groups.

For example, a tag template to identify all objects of the color red would be defined as "color.red". The unique id of the tag group is given first ("color"), followed by the unique id of the tag itself ("red").

Wildcards

The simplest tag template specifies an explicit tag (e.g. "color.red"). However, the '?' character can be placed at the end of a comparison string to denote a wildcard. When a wildcard is used, the template will match all tags whose unique id matches the comparison string up to the '?'. For example, the tag template "color.bl?" would match both of the tags "black" and "blue" for the group "color", but it would not match the tag "red". The wildcard character can only be specified at the **end** of a comparison string. Any characters occurring after the '?' will trigger a compiler error as invalid syntax.

Some examples of tag templates are provided below to illustrate how they can be used.

color.red	Matches only the tag "red" within the group "color".
color.bl?	Matches any tag that starts with the letters "bl" within the group "color". For example, this template would match tags for the colors "black" and "blue", but it would not match "red". It would also match the tag "bl".
color.?	Matches any tag within the group "color", provided the object has at least one "color" tag. If the object does not possess any tags from the group "color", the template will resolve to "false".

Important Considerations

Keep the following issues in mind when dealing with tag templates.

- All tag ids are case-specific, and so are tag templates. Be careful to specify the correct case for all templates.
- The comparison string utilizes the unique id for tags. If there are a number of related tags with which you want to be able to utilize wildcards, consider assigning them unique ids that make this possible. For example, if you have a "bigorc" and a "smallorc" tag, you could rename them as "orcbig" and "orcsmall". Using wildcards, you could reference both tags via the comparison string "orc?". This naming technique allows you to reference the two "orc" tags either independently or collectively, as the situation dictates.
- The tag template "group.?" can be extremely useful in various situations. Table selection and validation rules are often specific

to objects from a given category. By creating a tag group for the desired category and using the "group.?" syntax, you can instantly identify all objects from a particular category of interest.

Category: Basic Concepts and Terminology

Tag Terms

Context: HL Kit ... Basic Concepts and Terminology ... Tags and Tag Expressions

Contents

Overview

Tag expressions are built upon individual tag terms. Within a tag expression, every term is evaluated against the tags assigned to an object. Terms can either be "simple" terms or "arithmetic comparison" terms. Every individual term yields a result of either "true" or "false", allowing the terms to be combined into more complex Boolean expressions.

Simple Terms

Simple terms are just that – simple. A simple term must be one of the following:

TRUE	The literal value "TRUE" can be specified (case **is** important).
FALSE	The literal value "FALSE" can be specified (case **is** important).
tag template	A tag template can be specified, with the term returning a Boolean result of true or false. When a tag template is evaluated as a simple term, the result is true if the object contains one or more tags that match the tag template and false if no tags match the template.

Arithmetic Comparison Terms

There are times when you will need to identify objects which satisfy criteria that don't simply yield a "match" or "no match" result. In these instances, the object is required to comply with numeric constraints associated with the tags assigned. For example, a rule might pertain only to spells with a level of 5 or more. In situations like this, it's not appropriate to test for all the possible values (e.g. for the proposed sample rule, you would not want to test separately for spells of levels 5, 6, 7, 8, etc.). To handle special relationships like this, tag terms can be defined using arithmetic comparisons.

Arithmetic comparison terms always take the traditional form "left oper right", where "oper" is one of the arithmetic comparison operators: '<', '<=', '>', '>=', '=', or '<>' (the last one indicating "not equal"). Both "left" and "right" must evaluate to numeric values, with the comparison term yielding a "true" or "false" result based on the relationship between the values for "left" and "right". In this type of usage, the left and right sides of the comparison must be one of the following:

integer	A literal integer value can be specified (e.g. "1234"). Use of a literal integer value is restricted to the **right** side of an arithmetic comparison term only. An error occurs if a literal integer value is used on the left side. The integer value may **not** be negative.
count:template	A standard tag template of the form "group.template", subscribing to the syntax set forth previously (e.g. "color.bl?"). The value generated is the number of tags that have been assigned to the object which match the template (wildcards are allowed per standard tag template syntax).
low:template	All tags that match the specified template are identified, and an integer value is extracted from the end of each tag. The lowest (i.e. minimum) value of all tags is the value yielded. Integer value extraction is described below. Example: "low:cost.gold?".
high:template	Identical to "low:template", except that the highest (i.e. maximum) value of all matching tags is yielded. Integer extraction is described below. Example: "high:cost.gold?".
val:template	The first tag that matches the specified template is identified, and an integer value is extracted from that tag and returned. Integer value extraction is described below. Example: "val:cost.gold?". **NOTE!** If the template matches multiple tags, there is no guarantee which tag will be retrieved, so this form should only be used when you know there is a single tag matching the template assigned to the object. Using this form is much more efficient that the "low" and "high" forms, so make use of it whenever it is safe to do so.

NOTE! The prefix may optionally be omitted from a tag term within an arithmetic expression. If no prefix is specified, then the default prefix of "count" is assumed. Therefore, the term "color.blue" is equivalent to "count:color.blue" within an arithmetic comparison term, such as below.

```
(color.blue >= 1)
```

Tag Value Extraction

Every tag can be converted to an integer value. Extracting an integer value from a tag utilizes the unique id of the tag and not its name. Starting at the **end** of the tag's unique id, all digits are extracted until a non-digit is encountered. Those digits are then converted to an integer value. For example, integer value extraction from the tag "blue1234" would yield the value "1234". Similarly, the tag "blue5x123" would yield the value "123".

If a tag does not end in any digits, then the tag will yield "no" value, which will be treated as a value of zero when an integer value is required for the tag. For example, the tag "blue567xyz" would yield no value, since only the trailing digits are used and all earlier digits are ignored.

NOTE! Depending on the situation, there may be a critical distinction between a tag with "no" value and a tag with a "zero" value.

Be sure to keep the following issues in mind when dealing with tag values:

- Negative values are never extracted from tags, since only digits are considered when extracting the value.
- Under some circumstances, a tag will have no value and be referenced as if it has a value. Tags with "no" value are ignored in those situations (e.g. within summations and averages), while a tag with a "zero" value is processed with that value.
- An object that does not have any tags matching the specified template and for which a value is referenced will **never** satisfy an arithmetic comparison under any circumstances. For example, the comparison "val:attack.? < 3" will always fail for any object that has **no** tag from the "attack" group. Similarly, the comparison "val:attack.? >= 3" will also fail for those same objects. If an object has no value, it can never be compliant with a value-based comparison.

Field Value Tests

There are times when tracking a piece of information for an object really needs to be handled via a field value, yet you still need to test that state within tag expressions. One option would be to maintain both the field value and a corresponding tag that can be tested in a tag expression, but that approach quickly becomes difficult to maintain. To alleviate this, the Kit allows direct access to field values from within tag expressions.

You can reference field values using the syntax "fieldval:fieldid", where "fieldid" is replaced by the unique id of the field to access. Other than that, you can utilize field references just like any other arithmetic comparison term within the tag expression. For example, the tag expression "fieldval:strength < 10" would be satisfied if the object possesses a "strength" field and that field has a value of less than 10.

Only the fields for the appropriate object context can be referenced via tag expressions. Consequently, if the tag expression is being applied to a pick, only the fields defined for that pick can be referenced. If a field reference is used within a tag expression that either is not a pick/thing context or does not contain the specified field, a run-time error is reported to the user. If a tag expression will be used against things/picks that derive from different component sets and may not all possess a particular field, the author can include a test for the needed component within the tag expression to verify the presence of the field before accessing it. For example, if the field "foo" is defined within the "compid" component, the following syntax can be used to safely test the field value:

```
(component.compid & (fieldval:foo >= 1))
```

Be sure to keep the following important issues in mind when utilizing field values via tag expressions:

- Since only things and picks possess fields, access to field values will only work when the target object is a thing or a pick. There are times this might not be immediately obvious. For example, you might assume that field values can be referenced within a thing "condition" tag expression, except that the "condition" test is applied against the tags of the container, which means there is no thing/pick context to retrieve any fields from.
- In addition to a variety of standard tag expressions within a thing/pick context, access to field values can also be utilized from within most scripts. Any time that a pick target context is accessed from within a script, the "tagexpr[...]" target reference can safely utilize a field value references against the fields within that context.
- Do NOT use field value access as a crutch within tag expressions. Field value access from tag expressions is significantly more expensive to process than testing a tag, so use tags whenever possible and only use field values when absolutely necessary.

Explicit Scope Restrictions

While not common, situations will arise where you'll want to test the tags of a thing/pick and the tags of the hero to determine whether a tag expression should be satisfied. For example, consider a situation where an assortment of things is only available if the actor belongs to a particular group of classes. Assuming those classes assign an identifiable tag to the actor, you could filter the things displayed based on the presence of appropriate tags on the things and the tag on the actor.

To support these situations, any tag template used within a tag expression can specify a prefix of "hero#" on the template. When this prefix is assigned, the tag template is compared directly against the tags of the containing hero context, even if a pick or a child container is the established context. For this to work, the prefix must appear immediately prior to the tag template, such as in the examples below.

```
group.tag & hero#group.tag
```

OR

```
val:hero#group.tag
```

NOTE! Unlike with scripts, you **cannot** transition through the hierarchy within the tag terms. You can test tags on the current context or directly on the hero. Those are the only options, and they will be all you need 99.9% of the time.

Category: Basic Concepts and Terminology

Tag Expressions

Context: HL Kit … Basic Concepts and Terminology … Tags and Tag Expressions

Overview

Combinations of tags indicate meta-characteristics of an object. Tag expressions are the mechanism through which objects with specific combinations of attributes can be identified. Specifically, tag expressions are Boolean expressions that delineate a combination of tags that an object must possess. For example, a tag expression could easily be used to determine if an object is a "first-level wizard spell". Separate tag groups could be defined that identify the "class" of the object (fighter, wizard, etc.) and the "level" of the object (1st, 2nd, etc.). An additional tag group could be defined that identifies the "type" of the object (e.g. spell, weapon, skill, etc.). A tag expression could then be written to identify all objects that possess tags for all three characteristics: the "class.wizard" tag, the "level.1" tag, and the "type.spell" tag.

Syntax

Tag expressions are Boolean expressions comprised of tag terms. The tag expression evaluates each of the tag terms against a particular object to determine whether the object does or does not match the criteria. These terms are then combined using Boolean logic, where the expression specifies the relationships to use. Four Boolean operators are supported, and parentheses may be freely used to define appropriate groupings of sub-expressions. The operators supported are outlined below.

&	Logical "And". Evaluates the two terms on either side and returns "true" only if both terms are "true". Example: "term1 & term2".
|	Logical "Or". Evaluates the two terms on either side and returns "true" if either of the terms is "true". Example: "term1 | term2".
^	Logical "Xor". Evaluates the two terms on either side and returns "true" if exactly one of the terms is "true". Example: "term1 ^ term2".
!	Logical "Not". Evaluates the following term and returns "true" if that term is "false". Otherwise, it returns "false". Example: "!term1".

Examples

The following are a couple of sample expressions utilizing parenthetical sub-expressions and combinations of the above operators.

```
(TRUE & FALSE) ^ (TRUE & TRUE) ^ (FALSE & FALSE)
```

The above example evaluates to "true". The first term evaluates to "false" and the second term evaluates to "true". The Xor of "true" and "false" (the first two terms) yields "true". The third term evaluates to "false". The Xor of "true" and "false" (the result of the first Xor and the third term) yields "true".

```
TRUE ^ !(FALSE | TRUE)
```

The above example evaluates to "true". The sub-expression "(FALSE | TRUE)" evaluates to "true". The negation of the sub-expression yields "false". The Xor of "true" and the negated sub-expression result of "false" yields "true".

Category: Basic Concepts and Terminology

Arithmetic Expressions Within Tag Expressions

Overview

There will be times when you will need more than just a simple total of objects that match a set of criteria. For example, what if you need to calculate the combined total between two different groups of objects? Or the ratio between two groups of objects? In these situations, you will need to utilize an arithmetic expression to perform the calculation desired.

Usage and Syntax

Arithmetic expressions utilize the same basic syntax you learned back in elementary school. Expressions are formed out of numeric values that are combined using the standard arithmetic operators: '+', '–', '*', and '/'. An additional operator, %, can also be used that yields the modulus (i.e. remainder) after division (e.g. "5 % 3" would yield "2"). Arithmetic expressions can utilize parentheses to dictate how evaluation should be performed, with standard rules for arithmetic being used when no parentheses are specified (i.e. multiplication and division take precedence over addition and subtraction).

The numeric values within an arithmetic expression must be either a literal value of some sort (e.g. "3") or a standard tag term suitable for the context that yields a value. For example, within the context of a rule, tag terms can use the syntax "val:component.spell?". This same term could be used freely within an arithmetic expression that is utilized within a rule.

The following is an example tag expression that incorporates an arithmetic expression within it. In this example, the tag "color.red" must be defined, the tag "move.fast" must be defined, and the combined values of the "attack.?" and "defense.?" tags must be no more than 5.

```
color.red & (val:attack.? + val:defense.? <= 5) & move.fast
```

Important Considerations

Be sure to keep the following issues in mind when dealing with arithmetic expressions.

- Arithmetic expressions may contain any number of terms within them. However, a maximum of 20 special terms may be utilized within a single arithmetic expression. A special term is any non-literal value (e.g. "val:attack.?"). This probably seems like an absurdly high number for any typical use for a role-playing game, but the limit needs to be specified for completeness. If you define an arithmetic expression that contains more than 20 special terms, the expression will fail to compile and be reported as containing a syntax error.

Tag Expression Types

Context: HL Kit ... Kit Reference

The Kit leverages a variety of tag expressions for different purposes. The topics below provide a brief discussion of both the role and behavior of each different type of tag expression.

- Live Tag Expression
- Container Tag Expression
- Match Tag Expression
- List Tag Expression
- Candidate Tag Expression
- Restriction Tag Expression
- Secondary Tag Expression
- Existence Tag Expression
- HoldLimit Tag Expression

Category: Kit Reference

Live Tag Expression

Context: HL Kit ... Kit Reference ... Tag Expression Types

The role of the Live tag expression is to determine whether a visual element should be considered "live". When a visual element is "live", it is fully operational within the interface. When non-live, a visual element is treated as if it does not exist, being omitted from display and not processed in any way. All attempts to manipulate a non-live visual element via scripts are simply ignored.

The Live tag expression is always applied against the container whose information is to be displayed within the visual element. This will typically be the currently active actor, although it will sometimes be a gizmo (when editing a gizmo) or a different actor (when displaying material in places like the Dashboard). The tag expression is compared against all of the tags possessed by the container at the conclusion of the evaluation cycle, so the effects of all dynamically assigned and deleted tags are included. If the tag expression is satisfied, the visual element is considered live and shown normally. Otherwise, the visual element is deemed non-live and hidden from display.

The Live tag expression is applied anew after every evaluation cycle. Consequently, it is possible to have visual elements dynamically appear and disappear in response to user actions. This can range from a simple portal changing state to an entire tab panel or summary panel transitioning its state.

Category: Kit Reference

Container Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Container tag expression is to determine whether a structural element should be considered "live". When a structural element is "live", it is fully operational within over data hierarchy. When non-live, a structural element is treated as if it does not exist, so it is never shown to the user and no tasks associated with the element are processed in any way. All attempts to access or manipulate a non-live structure element via scripts will either be ignored or result in a run-time error.

The Container tag expression is always applied against the container of the structural element. In the case of a thing that has not yet been added to the portfolio, the tag expression is applied against the prospective container to which the thing will be added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. If the tag expression is satisfied, the structural element is considered live and behaves normally. Otherwise, the structural element is deemed non-live and its effects are ignored.

If the test occurs during the evaluation cycle, the test is applied against the current set of tags for the container. Consequently, it is critical that you pay attention to timing considerations and ensure that all the necessary tags are manipulated on the container prior to evaluating the tag expression.

The Container tag expression is applied whenever it is triggered. If that happens to be during the evaluation cycle, then it is applied during every evaluation cycle. This means it is possible to have structural elements dynamically change their live state in response to user actions. This in turn can cause structural elements to appear and disappear within the character, depending on how you setup your data files.

Category: Kit Reference

Match Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Match tag expression is to determine whether a thing should be included in a set of elements for processing in some way. For example, an Eval script on a component will use a Match tag expression to identify the subsets of things to which the script should be applied. The implications of the Match tag expression are clear and simple, since the thing is either included for processing or not.

The Match tag expression is always applied against the thing that may be processed. The tag expression is compared against all of the tags possessed by the thing. Since tags possess a fixed set of tags, a given tag will always either satisfy a Match tag expression or not. If the tag expression is satisfied, the thing is included in the processing it is being considered for. Otherwise, the thing is omitted from processing.

Category: Kit Reference

List Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the List tag expression is to determine whether a pick or thing should be included in the list of items shown for a table-based portal. If the tag expression is satisfied, then the pick/thing is shown within the table. Otherwise, the pick/thing is omitted from the table.

The List tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. Since populating tables is performed after the evaluation cycle completes, the tags used for picks are the final set of tags when evaluation completes.

The List tag expression is applied anew after every evaluation cycle. Consequently, picks that dynamically change their state can appear and disappear within tables based on changes in how they satisfy the tag expression.

Any item that is added to the character via a given portal is **always** shown within that portal. This overrides the behavior of the List tag expression. Even if the object does not satisfy the List tag expression, it will be shown in the portal if it was previously added via the portal. This ensures that picks/things added via a portal are always made visible so that they can be readily deleted by the user.

Category: Kit Reference

Candidate Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Candidate tag expression is to determine whether a pick or thing should be included in a choose form. The choose form contains the list of picks/things from which the user can select when adding items to a portal (i.e. dynamic table, chooser, or thing-based menu). If the tag expression is satisfied, then the pick/thing is shown within the list of options and can be added by the user. Otherwise, the pick/thing is omitted from the available list.

The Candidate tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. The Candidate tag expression is only invoked when the user triggers the portal to display the list of options to choose from. Consequently, the tags used for picks are the final set of tags after evaluation completes. This also means that picks can dynamically change their state and change whether they appear as valid choices within a given portal.

When a Candidate tag expression is used in conjunction with a List tag expression in a dynamic table, special behaviors can be leveraged. If you omit the Candidate tag expression entirely, the List tag expression is implicitly assumed, resulting in the user being able to select the same set of items that will be displayed. If you specify both tag expressions, the Candidate tag expression will normally supersede the List tag expression. However, if you specify the "inheritlist" attribute as "yes" on the Candidate tag expression, then the two tag expressions will be considered additive. Objects included in the selection list must then satisfy **both** the Candidate tag expression **and** the List tag expression. This allows you to avoid redundantly defining the logic of the List tag expression in both places while also ensuring that the items shown for selection correspond to the items that are intended to be shown in the table.

Category: Kit Reference

Restriction Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Restriction tag expression is to further limit the set of things or picks that are made available for selection through a portal. It is always used in conjunction with a Candidate tag expression and ensures that an item is only ever added to a specific **table** a single time. The Restriction tag expression is tested against all objects that are valid candidates for selection. If any of those objects also satisfies the Restriction tag expression **and** already exists within the table, they are precluded from being added to the table a second time and dropped from the selection list. This makes it possible to limit the user to only add an item once without having to designate the thing as unique.

The Restriction tag expression is always applied against the pick/thing itself. The tag expression is compared against all of the tags possessed by the object. For picks, this includes the effects of all dynamically assigned and deleted tags. The Restriction tag expression is only invoked when the user triggers the portal to display the list of options to choose from. Consequently, the tags used for picks are the final set of tags after evaluation completes. This also means that picks can dynamically change their state and change whether they appear as valid choices within a given portal.

An excellent example of where the Restriction tag expression comes in handy is with the d20 System. In d20, spells may be memorized multiple times, so the spells cannot be designated as unique. However, a wizard's spellbook will only possess a single instance of each spell the wizard knows. A Restriction tag expression on the spellbook table ensures that spells are unique within the spellbook.

Category: Kit Reference

Secondary Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Secondary tag expression is very different from most other tag expressions. It is always used in conjunction with a portal that adds a pick to a container, such as a dynamic table or chooser. Instead of being processed as part of the portal, the tag expression is attached to the pick. When the pick is selected and added to the container, the Secondary tag expression is also added to the new pick. At that point, the tag expression becomes part of the pick for the rest of the pick's existence.

Once added to the pick, the Secondary tag expression behaves just like a Container tag expression. In fact, the name of the mechanism derives from its use, as it becomes a secondary container tag expression. The Secondary tag expression must be satisfied in order for the pick to remain "live". If the tag expression stops being satisfied, the pick becomes non-live and is treated as if it no longer exists within the container.

The Secondary tag expression is always applied against the container to which the pick was added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. The Secondary tag expression is invoked every evaluation cycle at the timing specified. This means that the pick can transition between live and non-live in dynamic fashion, depending on changes that occur with the container.

It's also valid to omit the timing for a Secondary tag expression and use default timing. The default timing is Setup/5000. However, you can setup whatever you want as the default timing for the Secondary tag expression within the definition file.

As a concrete example, consider the D&D 4th Edition game system. If the character selects the appropriate feats to multi-class, he may then choose powers from the new class. However, after those powers are added, what happens if the user deletes the multi-class feat? The separate table for selecting the additional powers disappears, so there is no way for the user to delete the powers. The easy solution is to assign a Secondary tag expression to the table of multi-class powers that ensures all powers added require that the character be multi-classed. This way, if the multi-class feat is deleted, all of those powers that rely on the feat will fail the Secondary tag expression and be treated as if they were never selected.

Category: Kit Reference

Existence Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the Existence tag expression is very similar to the Secondary tag expression. It is always used in conjunction with a portal that adds a pick to a container, such as a dynamic table or chooser. Instead of being processed as part of the portal, the tag expression is attached to the pick. When the pick is selected and added to the container, the Existence tag expression is also added to the new pick. At that point, the tag expression becomes part of the pick for the rest of the pick's existence.

Once added to the pick, the Existence tag expression governs whether the pick continues to exist within its container (hence the name). The Existence tag expression must be satisfied in order for the pick to continue its existence and avoid being automatically deleted. If the tag expression stops being satisfied, the pick is immediately deleted from the container, just as if the user had deleted it. This is a much more severe action than the Secondary tag expression, which simply causes the pick to go non-live. However, it is sometimes the better choice.

The Existence tag expression is always applied against the container to which the pick was added. The tag expression is compared against all of the tags possessed by the container, which includes the effects of all dynamically assigned and deleted tags. The Existence tag expression is invoked every evaluation cycle at the timing specified. This means that the pick can fail the tag expression at any time, depending on changes that occur with the container. Once that occurs, the pick is automatically deleted in its entirety.

It's also valid to omit the timing for an Existence tag expression and use default timing. The default timing is Setup/5000. However, you can setup whatever you want as the default timing for the Existence tag expression within the definition file.

As an example, consider the wizard class within the d20 System. When a level of wizard is added, the user can add all sorts of spells to the wizard, both within the spell book and as memorized spells. Once those spells are added, they are independent of the wizard class level pick. This means that, if the user deletes the wizard level, all of the spells will still exist on the character. You could use a Secondary tag expression for this, but there would be no way for the user to delete these spells if the wizard class tab disappears. It would also be a real nuisance for the user to delete all of the spells, one at a time. If you use an Existence tag expression on the spells, then they will all be instantly discarded when the user delete the last wizard class level pick.

Category: Kit Reference

HoldLimit Tag Expression

Context: HL Kit … Kit Reference … Tag Expression Types

The role of the HoldLimit tag expression is to determine whether a piece of gear can be held by another piece of gear. By default, any piece of gear can be held within gear designated as a "holder". With the HoldLimit tag expression, the set of gear that can be held by a holder is further restricted. Only a piece of gear that also satisfies the HoldLimit tag expression is allowed to be held by the holder.

The primary use of the HoldLimit tag expression is to leverage the containment mechanism to associate weapon options with the various weapons. For example, a laser sight might confer bonuses to the use of a gun, but that laser sight can be moved between different weapons. Through the use of the HoldLimit tag expression, the gun can be restricted to only hold laser sights and other similar weapon options. The user can then move the laser sight gear from one gun to another, with a script being used on the laser sight to automatically apply the appropriate bonus to the gun the laser sight is attached to.

The HoldLimit tag expression is only applied against tags that are assigned directly to a thing at definition, including any component tags. Any tags that are assigned dynamically after the gear is added to the character are ignored. Consequently, the tag expression is evaluated against each piece of gear when the user brings up the menu to move the gear to a new holder. Once a piece of gear is assigned to a holder, no further evaluation is performed, unless the user chooses to move the gear again.

Category: Kit Reference

Scripting Language Overview

Context: HL Kit … Kit Reference

The scripting engine is a fundamental element of the Kit and you will find yourself writing numerous scripts to successfully manage character data and present it to the user. The Scripting Language section provides in-depth details on how to utilize the scripting language for your data files, and authors should be familiar with all the concepts presented herein before trying to write any scripts. The specifics of individual scripts and details of accessing the information within particular objects will be found in the sections that follow this one.

The scripting language documentation assumes that readers have a minimal familiarity with programming concepts (e.g. the notion of variables). As such, this documentation is not designed for a person with zero understanding of programming. However, anyone that has picked up a programming book and spent a couple of days toying around with a programming language should have plenty of background to work with HL scripts.

IMPORTANT! The Scripting Language documentation assumes you are already familiar with the basics of scripting and the evaluation cycle. If you have not already done so, please review the section Data Manipulation Basics before proceeding with these topics.

IMPORTANT! The scripting language and parsing mechanisms used by the Kit are relatively simple. This means that certain features provided in more complex programming languages are not available, and this is intentional. Complexity is not needed to support the writing of typically short and simple scripts. In addition, a simple language makes it relatively easy for non-programmers to modify existing scripts and/or write their own. All the intricacies of multiple, highly complex game systems have already been implemented using the scripting mechanisms provided within the Kit, so you should have everything you need to fully develop data files for whatever game system you set your sights on.

Category: Kit Reference

Language Syntax

The scripting language used by the Kit is not a clone of another language, although it does share similarities to a few. This section outlines the basic syntactic rules that govern the language.

The scripting language is line-based. Exactly one statement must be on a line, and each line is terminated by a carriage return (or newline). Long statements may **not** be split across multiple lines.

Any script line on which the very first non-whitespace character is a '~' is treated as a comment and ignored. Use of the '~' character anywhere else on a line (i.e. after the first non-whitespace character) is treated as normal scripting code. Therefore, it is not possible to specify "end-of-line" comments.

All script code is case-sensitive. Uppercase letters are distinct from lowercase characters in all circumstances. Therefore, the variable "foo" is distinct from the variables "Foo" and "FOO". This applies to all facets of the scripting language, which uses lowercase text exclusively for all keywords (e.g. "var", "if", "then", etc.).

Declaring Variables

Context: HL Kit … Kit Reference

Variables can be declared to store data for subsequent use within the script. Variables have the same naming rules as unique ids: maximum of 10 characters, limited to alphanumeric characters and the underscore, and cannot start with a digit.

Variables can be either a number (tracked as a floating point value internally) or a string of text. String variables have a maximum size limitation of 5000 characters, which should be more than adequate for any practical purpose.

Variables are declared with the syntax "var name as type", where "name" is the variable name to declare and "type" is the variable type. Valid values for the type are: "string" for strings and "number" for numeric values.

Examples of legal variable declarations are shown below:

```
var temp as number
var Word as string
var IN_CAPS as number
```

Examples of illegal or unwise variable declarations are shown here:

```
var name_too_long as number (over 10 character limit)
var one.two as string (illegal character used)
var 123456x as number (legal but unwise - starts with a digit)
var NUMBER as number (legal but unwise - don't rely on case to distinguish)
```

Within a script, variables are simply referenced using their name. Once a specific variable is declared to be of a particular type, it cannot be declared as another type. However, it is valid for the variable "foo" to be declared multiple times with the same type, as long as subsequent declarations are of the same type. If the variable already exists with the same type, the new declaration is simply ignored. This means that the following code is perfectly legal (though redundant):

```
var foo as number
var foo as number
```

However, the next block of code is not legal:

```
var foo as number
var foo as string
```

Every time a script is evaluated, it starts with a "clean slate". Therefore, values of variables do not persist from one invocation to the next. Once a variable is declared, it is initialized with a default value. For numeric variables, the default value is zero. For strings, the default value is an empty string ("").

Variables can be declared at any time within a script. The only requirement is that a variable must be declared before it is actually used. It is perfectly valid to declare a variable immediately before it is first used.

Category: Kit Reference

Basic Language Mechanisms

Context: HL Kit ... Kit Reference

The scripting language supports the following basic language mechanisms:

Assignment	Variables and attributes can be assigned values from other variables, attributes, and literal values. Assignment statements typically take the traditional form "x = y".
Numeric Assignment	Four additional assignment behaviors are supported that are exclusive to numeric values. The first two are the increment ("+=") assignment and decrement ("–=") assignment. Both are essentially a shorthand notation for the statements "x=x+y" and "x=x–y", respectively. Consequently, the statement "x+=y" is equivalent to "x=x+y". The other two are the multiply ("*=") assignment and divide ("/=") assignment, which are a shorthand for the statements "x=x*y" and "x=x/y", respectively. This assignment syntax can be incredibly convenient when identifiers are used with lengthy context transitions and target references.
String Assignment	One additional assignment behavior is supported that is exclusive to string values. It is the concatenate ("&=") assignment, which is essentially a shorthand notation for the statement "x=x&y". Consequently, the statement "x&=y" is equivalent to "x=x&y". This can be incredibly convenient when identifiers are used with lengthy context transitions and target references.
Arithmetic Expressions	Complex arithmetic expressions are fully supported by the scripting language. All of the standard operators are supported (+–*/), as is the use of parentheses to control the order of evaluation in an expression. An additional operator (%) is supported, which returns the remainder after dividing "x" by "y" (also called the "modulus").
String Expressions	Strings can be concatenated together through string expressions. To concatenate two strings, use the '&' operator (e.g. "str1 & str2"). No other string operators are supported.
Implicit Type Casting	The scripting language will automatically convert a variable or attribute between the two types when necessary. If you assign a numeric variable to a string, the value is converted to a string automatically (e.g. the value 521 is converted to the string "521"). Conversely, assigning a string variable to a number converts the string before the assignment takes place. However, going from a string to a number only converts the **leading** numeric portion of the string, so the string "1234hello789" is converted to the value 1234. IMPORTANT! Type casting only occurs with individual variables/attributes and not with entire expressions. Therefore, it is not valid to assign an arithmetic expression to a string, although it is valid to use a string variable within an arithmetic expression. If you want to assign an arithmetic expression to a string variable, you must first assign the expression result to a numeric variable and then assign that variable to the string. Similarly, it is valid to use a string variable as a parameter to an intrinsic function where a number is required, or vice versa.

The following sample code demonstrates the basic language mechanisms in use:

```
~Declare a variable and put a number into it.
var myint as number
myint = 7

~Declare another variable and set it equal to an arithmetic expression.
var temp as number
temp = (myint + 100) * 3

~Decrement temp by the value of "myint" times three.
temp -= myint * 3

~Set the value of a string to a numeric value, converting it to a string.
var mystr as string
mystr = temp

~Append the contents of two strings
mystr = "123" & mystr

~Append text to the end of the current string
mystr &= "123"

~Set a numeric variable to a string, automatically converting the result.
myint = mystr
```

Category: Kit Reference

Flow Control

Context: HL Kit ... Kit Reference

The scripting language supports a variety of basic flow control mechanisms, and each is described in the sections below.

Contents

Label and Goto Statements

The most primitive form of flow control is the use of "label" and "goto" statements. Label statements are defined with the form "name:", where "name" follows the same naming rules as variables. To transfer control to a label statement, a goto statement is used. The syntax for a goto statement is "goto name", where name corresponds to an existing label within the script. When a goto statement is executed, control passes to the label statement, and the next line executed in the script is the one immediately following the label statement. It is perfectly valid to issue a goto statement prior to the definition of the label statement, allowing you to skip over a section of the script if you wish. An example code block demonstrating the use of labels and gotos is given below.

```
var foo as number
foo = 10
goto mylabel
foo = 20
mylabel:
~The value of foo is still 10, since we skipped over the assignment to 20.
```

If/Then Blocks

A more traditional form of control flow is with the "if/then" block. Within an if/then block, a test of some sort is performed. Based on the results of that test, the following block of code may or may not be executed. The if/then block consists of two separate statements. The first statement is the test to be performed and uses the syntax "if (expr1 comparison expr2) then". The values of "expr1" and "expr2" can be any valid arithmetic expression. The valid values for "comparison" are the various relationship operators: '<', '<=', '=', '>=', '>', and '<>'. The first five of these comparison operators should be familiar, while the last one implies "not equal".

The "if" statement evaluates the two expressions and then compares them with the relationship indicated. If the comparison is satisfied, then execution continues with the line immediately following the "if" statement.

The second statement identifies the end of the if/then block and has the simple syntax "endif". If the comparison is failed, then execution skips over the if/then block until the "endif" statement is reached, at which point execution resumes. An example if/then block is shown below.

```
var foo as number
foo = 10
if (foo + 3 > 20) then
  foo = 1
  endif
~The value of foo is still 10, since we skipped over the assignment above.
```

If/Then/Else Blocks

The if/then block can be extended to be an if/then/else block. This form is useful when you need to execute one block of code if the condition is true and another block of code if it fails. The if/then/else block is identical to the if/then block, except that an additional "else" statement is inserted. This new statement demarcates where the block of code to execute on success ends and the block of code to execute on failure begins. If the conditional test is satisfied, then execution continues with the next line, but when the "else" statement is encountered, execution jumps to the line following the "endif" statement. If the conditional test fails, then execution jumps to the line following the "else" statement. An example if/then/else block is shown below.

```
var foo as number
foo = 10
if (foo + 3 > 20) then
  foo = 1
else
  foo = foo + 10
  endif
~The value of foo is now 20, since we executed the "else" clause only.
```

If/Then/Elseif/Else Blocks

To handle the situation where multiple value ranges need to be handled, the if/then/else block also supports an "elseif" statement. The "elseif" statement allows a separate condition to be tested, with its own block of code to be executed if the condition is satisfied. All condition blocks are tested in sequence, with only the first one that is satisfied having its code executed.

```
var foo as number
foo = 8
if (foo <= 5) then
  foo = 1
elseif (foo <= 10) then
  foo = 2
elseif (foo <= 15) then
  foo = 3
else
  foo = 4
  endif
~The value of foo is now 2, since we executed the first "elseif" clause only.
```

While/Loop Blocks

Another control flow mechanism provided by the scripting language is the "while/loop" block. The while/loop block executes a block of code repeatedly as long as a particular conditional test remains satisfied. The while/loop block begins with a statement of the form "while (expr1 comparison expr2)", where "expr1", "expr2", and "comparison" all behave identically to a standard "if" statement. The end of a while/loop block is identified by a "loop" statement, which consists solely of the keyword "loop" on a line. When the "loop" statement is encountered, the conditional test is evaluated again. As long as the condition remains satisfied, execution continues with the line following the "while" statement. Once the condition fails, execution jumps to the line following the "loop" statement. The loop will continue executing the code until the conditional test finally fails. An example while/loop block is shown below.

```
var foo as number
foo = 1
while (foo <= 10)
  foo = foo + 1
  loop
~The value of foo is now 11.
```

For/Next Blocks

The final control flow mechanism available is the "for/next" block. Like the while/loop block, the for/next block executes a block of code repeatedly, keying on the value of a variable (the "loop index"). The for/next block is identified by "for varname = expr1 to expr2", where "varname" is the name of the numeric variable to be used as the loop index and "expr1" and "expr2" are complex expressions. The variable is initialized with a given value and incremented by 1 every iteration of the loop. After the second value has been reached and processed, the iteration stops (so looping from 1 to 5 will run through the loop 5 times, the index taking on values of 1, 2, 3, 4 and 5 in succession before stopping). The loop index is automatically incremented by the for/next construct itself, and does not have to be maintained by the script writer.

```
var x as number
var text as string
text = "A list of some numbers: "
for x = 0 to (100 / 20 * 2)
  text = text & x
  next
~The string 'text' will now contain an ascending list of the numbers from 0 up to 10.
```

Nesting

All conditional statements (if/then, if/then/else, while/loop and for/next) can be safely nested within each other. For example, if two separate conditions must be satisfied in order to perform some action, you might have a script that looks similar to the example below.

When nesting conditional statements, each "endif" statement is associated with the closest "if/then" statement. Appropriate use of indentation can greatly enhance the readability of scripts when nested statements are employed.

```
if (x > y) then
  if (a > b) then
    z = 1
  else
    z = 0
    endif
else
  if (a > b) then
    z = 0
  else
    z = 1
    endif
  endif
```

IMPORTANT! The scripting mechanism is designed to handle scripts of only moderate size and complexity. As such, scripts possess a maximum nesting depth of 20 levels, which ought to be significantly more than any script should ever need.

If a script is too large or complex, an error will be reported when compiling the script. In general, a given script can utilize dozens of flow control constructs without becoming too complex, so the complexity limit should not be reached under normal conditions.

Done Statement

If you reach a point in the execution of a script where all necessary processing is completed, you can utilize the "done" statement to immediately exit the current script. Judicious use of "done" can simplify scripts and make them easier to maintain by eliminating a great deal of nesting and matching of if/then/else statements.

Two examples are shown below, where the second script accomplishes the same as the first. The key difference is that the second script utilizes the "done" statement. Use of the "done" statement is a matter of personal style, but it can be very handy for scripts with both simple and complex conditions.

```
if (foo <= 3) then
  ~do something
else
  ~lots of code
  if (for <= 6) then
    ~lots of code
  else
    ~lots of code
    endif
  ~lots of code
  endif
```

```
if (foo <= 3) then
  ~do something
  done
  endif
~lots of code
if (for <= 6) then
  ~lots of code
else
  ~lots of code
  endif
~lots of code
```

DoneIf Statement

There will be numerous situations when writing scripts where you'll want to test for a condition and exit out of the script. This amounts to writing an "if" statement and using "done" if the test is satisfied. That's three lines of code that you'll be using over and over again.

To simplify this, the scripting language provides a "doneif" statement. This statement takes a comparison expression and will exit the script if the test is satisfied, thereby reducing three lines of code to one. For example, the following two blocks of code are functionally equivalent.

```
if (foo <= 3) then
  done
  endif
```

```
doneif (foo <= 3)
```

Category: Kit Reference

Other Language Statements

Context: HL Kit … Kit Reference

The scripting language supports additional statements for special purposes that can be extremely useful.

Contents

Perform Statement

A large number of target references that apply changes to objects within scripts return a value that you can ignore. For example, the "assign" target reference that assigns a tag to an object always returns the value zero. Putting it to use will normally look like the following:

```
var result as number
result = assign[group.tag]
```

Declaring a variable in which to place a return value that you don't care about is an unnecessary nuisance. So the scripting language provides the "perform" statement. The perform statement tells the compiler to invoke the operation that follows the statement and simply throw away the value returned. The above example changes to the code below when the "perform" statement is employed.

```
perform assign[group.tag]
```

The net result is simpler, shorter, and clearer scripting code.

Debug Statements

If you need to write more than a few, simple scripts, the odds are that something won't work at some point along the way. So Hero Lab includes some very helpful debugging aids. One of these aids is the "debug" statement. The debug statement allows you to have Hero Lab output any information you can access via scripts to the screen during the evaluation process. When a debug statement is executed, the string you specify is evaluated and then output to the special debug information window within HL, where you can view the results and determine what changes need to be made to your scripts. An example code block using the debug statement is provided below.

```
var foo as number
foo = 10
var bar as string
bar = "hello"
debug "foo = " & foo & " and bar = [" & bar & "]"
```

The final output from the above script code will be as shown below within the debug info window.

```
foo = 10 and bar = [hello]
```

Using the debug statement requires that you utilize the info windows within HL. To view the debug output, go to the "Develop" menu within HL, select the "Floating Info Windows" option, and then select the "Show Debug Output" sub-option. This will display a window which contains the debug information that has been output from your scripts.

As your scripts continue executing and the debug window fills up, the oldest information will be lost off the top to keep memory consumption to a minimum.

Foreach Statement

There will be times when you need to iterate through a collection of objects, separately processing each individual object within the collection. For example, you might want to iterate through all the weapons possessed by the character to determine how many "hands" worth of weapons are equipped by the character.

In situations like this, the "foreach" statement provides the perfect solution. The foreach statement requires that you specify something to iterate over. The loop is terminated by the "nexteach" statement, resulting in the code within the foreach/nexteach block being invoked for every instance that satisfies the foreach criteria. The net result is that a typical use of a foreach/nexteach block looks like the code below.

```
foreach item in context where tagexpr sortas sortset
    ~code goes here
    nexteach
```

In the above code, the "foreach" statement stipulates what type of object is to be iterated ("item"), followed by the "in" designation and an appropriate context to search within ("context"). An optional tag expression can be specified that restricts the set of items iterated through by specifying the "where" clause and a string that contains the tag expression ("tagexpr"). The list of items can also be sorted via use of the "sortas" clause that specifies the unique id of the sort set to be used ("sortset").

There are multiple versions of the "foreach" statement that are supported. All versions work similarly, although each version iterates through a different assortment of items and context. The various versions are detailed in the sections below.

"foreach pick in container"

This form of "foreach" iterates through the picks that have been added to a container, processing only those that satisfy the specified set of criteria. With this version, the "context" must specify the container, which can be any valid context transition that identifies a container (e.g. "hero" or "child[pickid].gizmo"). The code within the "foreach" block is invoked for every pick in the container that satisfies the optional tag expression. Within the "foreach" block, the current pick being iterated can be accessed via the "eachpick." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach pick in hero where "group.tag"
    debug "id: " & eachpick.idstring
    nexteach
```

The above code will iterate through all picks within the hero and identify all that possess the "group.tag" tag. The "debug" statement (see below) will then output the unique id of each of the picks that are processed by the "foreach" statement, since the "eachpick." script context specifies access to the current pick within the body of the loop.

"foreach thing in component"

This form of "foreach" iterates through all things that derive from a specific component and satisfy any additional criteria. The "context" must specify the unique id of a component. The code within the "foreach" block is invoked for every thing derived from that component that also satisfies the optional tag expression. Within the "foreach" block, the current thing being iterated can be accessed via the "eachthing." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach thing in compid where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things derived from the component with the unique id "compid" and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

"foreach bootstrap in thing"

This form of "foreach" iterates through all things that are directly bootstrapped by a specific thing and satisfy any additional criteria. The "context" must specify the thing, which can be any valid context transition that identifies a thing or a pick. The code within the "foreach" block is invoked for every thing that is directly bootstrapped by the identified thing that also satisfies the optional tag expression. This includes both bootstraps defined on the thing and on any components the thing derives from. Within the "foreach" block, the current bootstrap thing being iterated can be accessed via the "eachthing." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach bootstrap in this where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things that are bootstrapped by the current thing/pick context ("this") and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

NOTE! This mechanism is primarily intended for use within scripts that synthesize description text for a thing.

"foreach bootstrap in entity"

This form of "foreach" iterates through all things that are directly bootstrapped by an entity and satisfy any additional criteria. The "context" must always be the literal string "entity", and it limits this statement to use within things that possess a child entity. The code within the "foreach" block is invoked for every thing that is directly bootstrapped by the child entity that also satisfies the optional tag expression. This includes both bootstraps defined on the entity definition and via the "child" element that attaches the entity. Within the "foreach" block, the current bootstrap thing being iterated can be accessed via the "eachthing." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach bootstrap in entity where "group.tag"
    debug "id: " & eachthing.idstring
    nexteach
```

The above code iterates through all things that are bootstrapped by the child entity of the current thing/pick context and that possess the "group.tag" tag. The "debug" statement outputs the unique id of each thing that is processed by the "foreach" statement.

NOTE! This mechanism is primarily intended for use within scripts that synthesize description text for a thing.

"foreach actor in portfolio"

This form of "foreach" iterates through the actors that exist within the portfolio, processing only those that satisfy the specified set of criteria. With this version, the "context" must always be the literal string "portfolio". The code within the "foreach" block is invoked for every actor in the portfolio that satisfies the optional tag expression. Within the "foreach" block, the current pick being iterated is always the "actor" pick for the current actor and can be accessed via the "eachpick." script context transition. The net result is that a typical use of this version of "foreach" looks like the code below.

```
foreach actor in portfolio where "group.tag"
    debug "name: " & eachpick.hero.name
    nexteach
```

The above code will iterate through all actors within the portfolio and identify all that possess the "group.tag" tag. The "debug" statement (see below) will then output the name of each actor that is processed by the "foreach" statement, since the "eachpick." script context specifies access to the current pick within the body of the loop and the parent actor is then used.

NOTE! This mechanism is primarily intended for use within the Dashboard and Tactical Console, where all the actors must be managed appropriately.

Notify Statement

In order to facilitate debugging and to provide a convenient means for data files to report special events to the user, the "notify" statement is provided. The notify statement works similarly to the "debug" statement, except that the resulting string is reported directly to the user via a message alert. For all practical purposes, the syntax for the "notify" statement is identical to the "debug" statement, except that the line starts with "notify", as shown below.

```
notify "This is the message displayed"
```

The notification is queued and then reported to the user at the first reasonable opportunity. This means that the message is not necessarily reported to the user immediately when it is triggered. Similarly, if there are multiple notification messages, they will be accumulated by HL and then reported collectively at the next opportunity. The messages will be displayed within a single report to the user, with each message on a separate line.

Append Statement

The "append" statement is used exclusively with the Synthesize script. When generating dossier output, the volume of text can be quite long, in which case the traditional "@text" special symbol becomes impractical and inefficient. So the append statement is provided as a means to output a chunk of text to HL and then allow the author to stop worrying about it. Once text is output via the append statement, it will be part of the synthesized output, and it will appear in the sequence it is output.

The syntax for the append statement is simple, as it uses a single string. When an append statement is executed, the string you specify is evaluated and then output as part of the dossier. This means that three append statements in a row will procedure output that is the concatenated result of all three strings, as shown below.

```
append "line #1" & @newline
append @boldon & "line #2" & @boldoff & @newline
append "line #3" & @newline
```

The final output that results from the above three statements would look like the following when synthesized for HTML:

```
line #1<br>
<b>line #2</b><br>
line #3<br>
```

Trustme Statement

As a general rule, scripts should not be directly modifying the contents of "user" fields. Those fields are intended for access only by the user, so attempts to modify them via scripts are inherently considered an error by the compiler. However, as with any rule, there are always the exceptions.

When a script needs to modify the value of a user field, the "trustme" statement can be used. This tells the compiler that you know what you're doing and can be trusted to not do things inappropriately. Once specified within a script, the script becomes trusted and the compiler stops checking for code that modifies user fields.

The syntax of the trustme statement is trivial. Simply place the statement on its own line within the script. There are no parameters, so it looks like below.

```
trustme
```

IMPORTANT! Whenever you think that you need to utilize the trustme statement, look closely at your script and what you're trying to do. It is rare that you should actually need to use trustme, and your data files probably can be better structured to eliminate the need for directly modifying a user field.

IMPORTANT! The following script types are automatically designated as "trusted". This is because their general nature is such that you'll often be needing to modify user fields.

- Trigger Script
- Creation Script
- TransactSetup Script
- TransactBuy Script
- TransactSell Script
- Merge Script
- Split Script
- NewCombat Script
- NewTurn Script
- Integrate Script
- Initiative Script
- LoadFixup Script

Category: Kit Reference

Language Intrinsics

Context: HL Kit ... Kit Reference

Overview

The scripting language has an assortment of intrinsic (i.e. built-in) functions that can be used for various purposes. Some intrinsic functions operate on strings, while others operate on numbers. Some intrinsic functions return strings, while others return numbers.

The purpose of intrinsic functions is to provide authors with re-usable mechanisms that can be used to perform script operations that arise on a recurring basis. For example, the Kit includes intrinsic functions for searching string, carving up strings, comparing strings, and replacing text within strings. Intrinsics are also included to determine the minimum or maximum of two values, round a number off at a certain level of precision, and generate a random number.

Intrinsic Functions

The complete list of intrinsic functions provided by the language is presented below.

length	number length(string str) Returns the number of characters in the string *str*.
left	string left(string str, number num) Returns a new string containing the leftmost *num* characters of the string *str*.
right	string right(string str, number num) Returns a new string containing the rightmost *num* characters of the string *str*.
mid	string mid(string str, number start, number num) Returns a new string containing a substring of string *str*. The new string begins with the character at position *start* and is *num* characters in length. Character positions are 0-based, so the first character in a string is at position zero (0).
pos	number pos(string str, string search) Return the character position where the string *search* first appears within *str*. Character positions are 0-based, so the first character in a string is at position zero (0). If search does not exist within str, a value of –1 is returned.
uppercase	string uppercase(string str) Returns a new string that is an upper case version of string *str*.
lowercase	string lowercase(string str) Returns a new string that is an lower case version of string *str*.
lastpos	number lastpos(string str, string search) Return the character position where the string *search* **last** appears within *str*. Character positions are 0-based, so the first character in a string is at position zero (0). If search does not exist within str, a value of –1 is returned.
asc	number asc(string str) Returns the ASCII value of the first character of string *str*.
chr	string chr(number val) Returns a new string that consists of a single character, which has an ASCII value of *val*. For example, 'chr(10)' is the newline character.
compare	number compare(string str1, string str2) Compare the two strings *str1* and *str2*. If the strings are identical, the value 0 is returned. If *str1* would appear before *str2* in an alphabetical sort (based on the ASCII code of each character), a value less than 0 is returned. If *str1* would appear after *str2* in an alphabetical sort, a value greater than 0 is returned. Note that all uppercase characters are sorted before lowercase characters.
replace	string replace(string str, string match, string replace, number maxcount) Searches through the string *str* and replaces all instances of the string *match* with the string *replace*. A maximum number of replacements is given by *maxcount*, with a value of zero indicating that all matches should be replaced. The converted string is returned, with the original string left untouched.

int	number int(number val) Returns a value that represents the integer portion only of *val*. For example, the integer portion of the value –123.45 would be –123.
random	number random(number range) Returns a random integer value between zero and *range*–1.
minimum	number minimum(number val1, number val2) Returns the lower of the two values given.
maximum	number maximum(number val1, number val2) Returns the higher of the two values given.
power	number power(number val1, number val2) Returns *val1* raised to the power of *val2* (e.g. x^y). For example, "power(4,2)" would yield 4 squared, which is 16.
nthroot	number nthroot(number val, number nth) Returns the nth root of a number, where *val* is the value to get the root of and *nth* indicates the root to obtain. For example, "nthroot(16,2)" would yield the square root of 16, which is 4.
round	number round(number val, number dec, number dir) Returns the rounded value of *val*, where *dec* indicates the number of decimal places at which to perform the rounding and *dir* indicates how to perform the rounding. If *dir* is zero, normal rounding is performed (e.g. 0.5-0.99 round up to 1.0 and 0.01-0.49 round down to 0.0). If *dir* is positive, the value is always rounded up. If *dir* is negative, the value is always rounded down. For example, "round(4.36,1,0)" would yield 4.4 and "round(4.36,1,-1)" would yield 4.3.
signed	string signed(number val) Returns a string that represents the properly signed version of 'val', including a prefix of either '+' or '-'. For example, the signed version of the value 1.42 would be "+1.42", while the signed version of the value -6.23 would be "-6.23".
decimals	string decimals(number val, number dec) Returns a string that contains the rounded value of *val*, where *dec* indicates the number of decimal places at which to perform the rounding (normal rounding is always performed). In addition, if the value requires fewer decimal places than specified, zeroes are appended to bring the value to the full number of decimals. This intrinsic is ideal for formatting currency with a fixed number of decimals, so that "decimals(1.5,2) produces "1.50" instead of the normal "1.5".
bitwise_and	number bitwise_and(number val1, number val2) Returns the bit-wise "and" of *val1* and *val2*, treating both parameters as integer values for the operation. The bit-wise "and" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 only if both *val1* and *val2* have that bit set to 1. For example, "bitwise_and(14,7)" would yield 6, since the two parameters have binary representations of "1110" and "0111", which results in a value with a binary representation of "0110" (or 6).
bitwise_or	number bitwise_or(number val1, number val2) Returns the bit-wise "or" of *val1* and *val2*, treating both parameters as integer values for the operation. The bit-wise "or" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 if either *val1* or *val2* has that bit set to 1, or if both have the bit set to 1. For example, "bitwise_or(10,3)" would yield 12, since the two parameters have binary representations of "1010" and "0011", which results in a value with a binary representation of "1011" (or 12).
bitwise_xor	number bitwise_xor(number val1, number val2) Returns the bit-wise "xor" of *val1* and *val2*, treating both parameters as integer values for the operation. The bit-wise "xor" performs a comparison of the two values, one bit at a time, with the resulting value having a bit value of 1 only if one of *val1* and *val2* have that bit set to 1 - not both. For example, "bitwise_xor(14,7)" would yield 9, since the two parameters have binary representations of "1110" and "0111", which results in a value with a binary representation of "1001" (or 9).
	number bitwise_not(number val) Returns the bit-wise "not" of *val*, treating the parameter as an integer value for the operation. The bit-wise "not" processes the value, one bit at a time, with the resulting value having each bit possess the opposite value

bitwise_not	that it started with. This means that any bit with a value of 1 becomes 0, and vice versa. For example, "bitwise_not(9)" would yield 6, since the parameter has a binary representation of "1001", which results in a value with a binary representation of "0110" (or 6).
empty	number empty(string str) Returns a non-zero value if the string *str* has a length of zero characters, else a value of zero is returned. The "empty" intrinsic is significantly faster than using "length" and comparing it to zero.
plaintext	string plaintext(string str) Returns the string *str* with all encoded text stripped out. This provides a quick an easy mechanism for converting a string with encoded text (e.g. color usage) to its equivalent without any special formatting.
today	number today() Returns the current date in the proper format utilized by fields that store dates and times which users can edit via "editdate" portals. This provides a convenient mechanism for initializing journal entries and any other situations with the current date.

Examples

Simple examples of how to use each of the various intrinsic functions are provided below.

```
var str as string
var len as number
var piece as string
var temp as string
var val as number
var total as number
var NewLine as string
var tag_count as number
var link_count as number
str = "hello there world there"

~Get the length of the string, which is 23
len = length(str)

~Extract the first 5 character of the string, which is "hello"
piece = left(str,5)

~Extract the last 5 characters of the string, which is "there"
piece = right(str,5)

~Extract 5 characters from the string, starting at position 6, which is "there"
piece = mid(str,6,5)

~Locate the first occurrance of the string "there" within the string, which
~starts at position 6
val = pos(str,"there")

~Locate the last occurrance of the string "there" within the string, which
~starts at position 18
val = lastpos(str,"there")

~Convert a string to all uppercase, then to all lowercase
temp = uppercase(str)
temp = lowercase(str)

~Get the ASCII value of the first character of the string
val = asc(str)

~Create a string of one character, where that character is an 'A'
piece = chr(65)

~Create a string of one character, where that character is a newline
NewLine = chr(10)

~Compare the two strings, with the first string being less than the second
val = compare(str,"second")

~Replace all instances of "there" with "change"
temp = replace(str,"there","change",0)

~Extract the integer portion only of –123.45, which is –123
val = int(-123.45)
```

```
~Generate a random number between 0 and 9
val = random(10)

~Determine the lower and higher of two values
val = minimum(123.45,val)
val = maximum(123.45,val)

~Calculate 4 squared, which yields 16
val = power(4,2)

~Calculate the square root of 16, which yields 4
val = nthroot(16,2)

~Round the value 4.36 normally to one decimal place, which yields 4.4
val = round(4.36,1,0)

~Get the signed version of a value
temp = signed(-123.45)

~Convert a value to formatted currency
temp = "$" & decimals(1.5,2)

~Perform bitwise operations on two values
total = bitwise_and(14,7)
total = bitwise_or(10,3)
total = bitwise_xor(14,7)
total = bitwise_not(9)

~Determine if a string is empty
value = empty(str)
```

Category: Kit Reference

Special Symbols

Context: HL Kit … Kit Reference

Many scripts have special pieces of information that are either passed into the script from HL or that the script must specify for subsequent use by HL. For example, a script-based label portal has two such values. On entry, the script is informed whether it is being applied to either a thing or a pick, since that distinction is important for some scripts. On exit, the script must tell HL what the final text is that should be displayed within the label.

Whenever a value needs to be passed into or out of a script, a "special symbol" is used. Special symbols behave exactly like variables, so they can be used just like a variable within the script. The name of each special symbol is always prefixed with the '@' character, which is followed by the name of the special symbol. For example, to access the special symbol "value", the syntax would be "@value".

The set of special symbols varies for each script, as will their use, with some scripts having zero special symbols and a few having many. The exact symbols for a given script are specified with the details for each script within in Kit Reference documentation.

Category: Kit Reference

Script Macros

Context: HL Kit … Kit Reference

The scripting language syntax is designed to support accessing a complex assortment of game elements. This can result in some rather lengthy terms to specify exactly the information you want to access. Many of these terms will be used extensively within the data files for a given game system. To simplify scripts, the Kit allows you to define an assortment of script macros that serve as a shorthand notation for quickly accessing common elements.

For example, in the data files for the d20 System, there are numerous effects that apply bonuses to skills (e.g. feats). The syntax for accessing the bonus value for a given feat looks something similar to the code shown below (for more details, please refer to the Definition File Reference).

```
hero.childfound[kTumble].field[Bonus].value
```

If you had to type this in everywhere, there would be lots of opportunity for errors and it would be tedious to type each time. So the d20 System data files define a "skillbonus" macro that can be used in place of the above syntax. Using the "skillbonus" macro, the resulting syntax is simplified to the following:

```
#skillbonus[kTumble]
```

By using a script macro, the code becomes simpler to write, plus it becomes clearer regarding what it's doing. The above macro is clearly accessing the skill bonus value for the skill with unique id "kTumble". This is much clearer than the lengthy text to which the macro corresponds.

The syntax for using a script macro is that it always starts with a '#' character, followed by the name of the macro. Parameters for the macro are placed within square brackets and separated by commas (if there are more than one). When HL compiles the data files, it knows to replace the macro with the corresponding long syntax, as specified for the game system, allowing the compiler to generate the proper behavior.

The data file structure will vary greatly from one game system to the next, as will the commonly accessed information for each game system. As such, the specific set of macros defined for each game system will vary based on what's most appropriate that game system. To simplify getting started with your own data files, the starter data files provided with the Kit include a number of useful script macros that you can both put to use and refer to as examples when adding your own new macros.

Category: Kit Reference

Re-usable Procedures

Context: HL Kit … Kit Reference

Procedures provide the means to define script logic in a single location and re-use that logic within multiple scripts by calling the procedure from each of those scripts. Procedures are an extremely convenient tool and are used regularly for a variety of purposes.

There are a number of important considerations that come in to play when using procedures. Some are fundamental to the nature of procedures, while others are selectable, allowing you to choose between various capabilities to better tailor the procedure to your needs.

The topics below summarize each of these considerations.

Contents

Inherit Caller's Context

When invoked, procedures inherit the initial script context of their caller. For example, if a Description script calls a procedure, the pick or thing that is the initial context of the script is also the initial context within the procedure. There is one exception to this, which is discussed in the topic on "Any" Procedures (below).

No Parameters

Unlike most programming languages, procedures have no parameters. You cannot pass parameters to procedures in the way that traditional programming languages do. However, you **can** pass information between caller and procedure via the use of variables (see below).

Inherit Caller's Local Variables

Procedures inherit all local variables of the calling script. However, you must still declare each local variable within the procedure so that the compiler knows that you intend to use them. When that is done, the value of each variable will be inherited when the procedure is called and can be accessed normally via the variable within the procedure. Using local variables, you can pass information between the calling script and the procedure. Changes made to variables within the procedure will remain in effect upon return to the calling script, so you can use this technique to pass information both from the script into the procedure and back out from the procedure to the calling script upon return.

Script Type Vs. Context

All procedures must be assigned either a specific script type or a more general script context. This assignment dictates important behaviors for the procedure that impact which scripts can legally call the procedure. Scripts can only call procedures that are assigned a compatible classification. Either the script type must match exactly or the script context must match generally. For example, you cannot call a procedure with "container" context from within an Eval Script, since that script has a "pick" context.

There is a distinct advantage to designating a specific script type for a procedure. If a script calls a procedure with the same script type, then the procedure has full access to all of the special symbols defined for the script. The special symbols behave just like inherited variables and are shared back and forth. If a procedure does not share the identical script type, special symbols are not accessible.

There is a different advantage to using a more general script context for a procedure. By using a general script context, a procedure becomes much more versatile. The procedure can be used from multiple different scripts of different types. For many scripts, using a matching context is preferable, since the procedure can then be re-used by more scripts.

The complete list of script types and contexts available for procedures is specified within the Kit Reference documentation.

"Any" Procedures

There is a special type of procedure that can be called from absolutely **any** script. In order to make this possible, though, the procedure must make an additional concession to ensure compatability. The procedure possesses **no** initial script context. This is because the same procedure could be called from different scripts, where each calling script has a different initial context.

The implications of having no context are significant. Without an initial context, the procedure cannot perform transitions to anything based on an initial context. This means that the only things that can be done are to either operate upon variables (which are shared with the caller) or specify a "safe" context and transition from there. For example, the procedure is always free to start with the "hero" context and locate the specific information it needs, since there is always a hero that can be identified and used.

Calling Other Procedures

Procedures **are** allowed to call other procedures. All of the rules above apply to procedures that are called from other procedures. The same variable space is shared, all of the special symbols are shared, and all of the rules regarding script type and context apply equally to any procedure, even one that is not called directly from a script.

Category: Kit Reference

Debugging Mechanisms

Context: HL Kit … Kit Reference

Overview

The development of a fully functional set of data files for a game system involves a great many steps and there are bound to be a few problems along the way. To handle this, the Kit provides a variety of mechanisms to help you quickly identify problems and diagnose how to fix them. In order to facilitate the resolution of problems, HL includes a variety of built-in mechanisms that can be leveraged when issues arise. Since errors in software are traditionally called "bugs", we refer to these mechanisms as "debugging" aids and the overall process is commonly referred to as "debugging" your data files.

Topics

The topics below outline a number of incredibly useful tools and techniques for quickly figuring out what is going wrong within your data files. Familiarize yourself with these debugging mechanisms now, since you will find yourself putting them to use at various points in the development process.

- Using Info Windows
- Debug Output Via Scripts
- Script Timing Issues
- Establishing Timing Dependencies

Category: Kit Reference

Using Info Windows

Context: HL Kit … Kit Reference … Debugging Mechanisms

Whatever the game system, HL will be tracking a great deal of information. Portfolios contain multiple heroes and heroes can have child gizmos. Heroes and gizmos contain a large number of picks, and those picks will contain a healthy number of fields. Everything has a wide assortment tags. And don't forget the task list that tracks all of the different operations that are performed on all of these objects. Even if the game system is relatively simple, there will be lots of information, and the volume increases dramatically as the game system complexity goes up.

At any point in time during development, you'll be working on some subset of this information. After you make some changes to the data files, you'll be expecting everything to work. However, if it doesn't work, you'll want to understand why. This is usually achieved by looking at the appropriate subset of information within HL that you are manipulating to see what **is** happening, which can typically narrow down where the error is lurking.

To help out in these situations, HL provides a bunch of different ways to view the information that is being tracked internally. This is accomplished through "info windows". Each info window is a floating window that is separate from the main HL window and contains specific information about some facet of the current portfolio. Every time that a change is made to portfolio, the contents of each info window are updated, so you can see the direct effects of changes in real-time.

Info windows can be created at any time by going to the "Debug" menu, selecting the "Floating Info Windows" menu, and then choosing the appropriate option from the sub-menu that is presented. Once created, the info window persists until you close it. You can move the info window around and resize it as you deem appropriate.

Each type of info window contains different information that may prove useful in different situations. The table below offers a quick summary of the information provided by each info window.

Hero Tags	Displays a list of all the tags assigned to the active hero. This list includes tags from three different sources. First, any tags dictated by the user's selection of sources and/or rule sets are included. Second, tags that are dynamically assigned to the actor by scripts are included. Lastly, tags that are automatically assigned to minions are included (only if the hero is a minion).
Hero Fields	Displays a list of all the fields assigned to the active hero. Heroes don't technically possess fields, but every hero automatically contains a single "actor" pick that does possess fields. Since the "actor" pick is considered synonymous with the actor in a variety of situations (e.g. the Tactical Console and Dashboard), the fields within the "actor" pick are considered to be "actor fields" (or hero fields).
Task List (Active Hero)	The complete set of all tasks for the active hero are listed. The task list includes all actions that are scheduled to occur during the evaluation cycle, such as Eval Scripts, Eval Rules, Bounding scripts, Condition tests, and a host of other operations. Anything that is scheduled to be performed at a specific phase and priority has a corresponding task that will appear in this list.
Task List (Full)	This option shows the complete task list for all heroes within the context of the current lead. When minions are used and those minions have timing inter-dependencies with their master, the interplay of all the tasks for all the actors can be critical. In those situations, the full list can be used to help diagnose timing errors between different actors.
Selection List	Displays a list of all of the picks that have been added to the active hero. There are a variety of mechanisms within the Kit that control whether picks are added to an actor, such as bootstraps, condition tests, and live tests. This info window can help you to diagnose whether things are behaving as you intend.
Selection Tags	When you select this option, a list of all of the picks for the active hero is presented. You can select as many picks as you want, after which separate info windows will be created for each pick you specified. The info windows will contain a complete list of all the tags that are assigned to the pick at the conclusion of evaluation. If a tag is deleted at some point during evaluation, it will not appear in this list.
Selection Fields	When you select this option, a list of all of the picks for the active hero is presented. You can select as many picks as you want, after which separate info windows will be created for each pick you specified. The info windows will contain a complete list of all the fields and their values for the pick at the conclusion of evaluation.
Debug Output	This option opens up a window into which debugging output is written during evaluation. This topic is discussed in detail in the following topics.

Category: Kit Reference

Debug Output Via Scripts

Context: HL Kit … Kit Reference … Debugging Mechanisms

Overview

In most traditional programming languages, there is the age-old technique of inserting a "print" statement at various points in the code to display the state information. This technique is used to verify that everything is behaving correctly and to diagnose errors that surface. The Kit provides two related mechanisms that offer a similar solution.

The "notify" Statement

The first mechanism is the "notify" statement within the scripting language. This statement allows you to synthesize a string and then display it to the user. The string can be anything, such as the contents of fields, the results of an arithmetic expression, or even just a simple error message. So you can display whatever information you need to convey.

The notify mechanism displays an alert message to the user, requiring the user to dismiss the message before continuing. It works very similarly to the way run-time errors are handled. Consequently, the notify mechanism is ideal for reporting unexpected and/or error conditions that are encountered during evaluation. It is also useful for inserting a quick verification of a few values during testing of new functionality.

The "debug" Statement

The second mechanism is the "debug" statement within the scripting language. Like the notify mechanism, this statement allows you to synthesize a string for output, but the resulting text is not displayed directly to the user. Instead, the text is sent to the Debug Output window, which can be shown via the "Floating Info Windows" sub-menu of the "Debug" menu.

When the Debug Output window is visible, the use of a sequence of "debug" statements will send a progressive series of messages to the window. You can use this technique to monitor the flow of execution through all of your scripts and watch both the values of fields and the presence of tags at various points during the evaluation cycle. Every time the evaluation cycle is triggered, all of the debug messages are output again, so you can easily watch the effects of changing selections and values within the character through each new evaluation cycle.

This particular technique is probably the single most valuable method for quickly diagnosing where things are going wrong within your scripts. Once you know where things are going wrong, it's usually pretty easy to figure out why the error is occurring and put in the proper correction.

Category: Kit Reference

Script Timing Issues

Context: HL Kit … Kit Reference … Debugging Mechanisms

Contents

Overview

We'll start this section off by answering the obvious question. What's a timing issue?

The evaluation cycle controls the order in which everything is processed for each character, and it's up to the author to make sure that every task is assigned an appropriate phase and priority. A timing issue is the result of scheduling a task in the wrong sequence, which causes the evaluation cycle to yield the wrong final results.

It doesn't matter how experienced you are as an author, script timing issues will crop up if the game system has even a modicum of complexity. The trick is in being able to recognize when you are dealing with a timing issue. In general, timing issues are identified via a process of elimination, wherein the possibility of all other types of errors is systematically ruled out.

This section provides assorted tips on how to recognize and resolve timing issues.

Using the Task List

The most fundamental tool for solving timing issues is the task list, which is accessible via the "Debug" menu and the "Floating Info Windows" sub-menu. The task list, as its name suggests, presents a single list of all of the different tasks that are scheduled during the evaluation cycle. This list is often rather lengthy, since every pick and every field has its own distinct set of tasks.

The list is sorted in the order that the tasks will be evaluation by the HL engine. For each task, you'll see its phase, priority, and a brief description. This description should help in identifying exactly what the task is doing and what it is operating upon.

If you think you have a timing issue, the thing to do is check the task list. Make sure that each of the tasks you're working with is occurring in the proper sequence. If you're integrating new functionality that relies upon other tasks to setup state appropriately, be sure to check that your new tasks are occurring after the existing tasks have been invoked. Similarly, if your new tasks apply modifications that other tasks will rely upon, make sure to verify that sequencing as well.

Remember that there are two separate task lists. In general, you will want to utilize the task list for the active hero. This task list is perfect for verifying timing relationships between tasks within a single actor, which is normally what you'll be trying to do. The full list contains all the tasks for all actors, interspersed amongst each other. The only time you'll want to use this task list is when you have masters and minions that have inter-dependencies between each other, such as familiars in the d20 System, which have traits derived from their master and also confer bonuses back onto their master.

NOTE! The task list is the **only** way to diagnose timing issues that involve tag expressions, such as live states and condition tests. Using the task list is critical to verifying that important tag expressions are being evaluated appropriately relative to scripts and/or other tag expressions.

Debug Output Tracing

If a review of the task list doesn't reveal the source of the problem, another useful trick is to insert assorted trace information into your scripts. For quick checks, the "notify" statement is convenient. However, in most cases, you're going to find the "debug" statement is preferable.

By sprinkling "debug" statements in various places within scripts, you can achieve two goals. First, you can report important state information during the course of the evaluation cycle. For example, you can output field values and tag lists at various points to ascertain that everything matches the state you expect at each of these junctures.

Second, you can report the overall execution flow of scripts, allowing you to verify that ActionA is actually being triggered and that it's occurring before ActionB. For example, when a field value is set or a tag assigned, you can output a message that relays that fact, with the option of also including details about the new state.

The debug output window displays message in the exact sequence that they are called from within scripts. Consequently, what you see within the debug output window is an accurate reflection of what is actually going on within your data files.

Based on the debug output, you should be able to spot the problem. Sometimes, you'll start with only a handful of debug statements and then add more based on the output you see. This technique will systematically narrow down where the timing problem is occurring until you can finally pinpoint it and determine a proper fix.

Script Timing Output

There are times when you have numerous different tasks all working together for some purpose and a timing issue arises. In situations like this, it can be very tedious and time-consuming to pour through the task list and cross-reference everything to make sure all the timing relationships are correct. So the Kit provides a convenient way to check timing information without having to wade through the task list.

From within any script, you can use the target identifier "state.timing" to retrieve the timing information for that script. The string returned has the format "phase/priority", consisting of the name of the phase followed by the numeric priority at which the script is running. By using this within a "debug" statement, you can report the timing of the current script. For example, you could use a statement like the one below to report the timing when a field is modified and its new value.

```
debug "Script 'blah' at " & state.timing & " - Field 'foo': " & field[foo].value
```

When you have numerous scripts all working together, or when you have the same field being setup by multiple different scripts under different conditions, this technique can be extremely handy. Along with the timing, you can include information about the script that is being invoked, and the net result is debug output that tells a very clear and specific story about what changes are occurring from which sources and at which times. Armed with detailed output like this, you should have a relative easy time nailing down the problem.

Diagnosing a Timing Issue

Whenever you think you have a timing issue, the first diagnostic step is always to review each of the tasks involved in the behavior you're implementing. For each task, you need to double-check the associated tag expressions and/or script code to make sure everything should be working correctly.

Once you've verified that everything looks right, the next step is check the appropriate info windows to see whether all of your assumptions are being satisfied. Do fields contain the values they should? Are all the proper tags assigned where they should?

If everything appears correct, then you may very well have a timing issue. Why? Because the info windows only show what everything looks like at the **end** of the evaluation cycle. If TaskA is evaluated before TaskB, and TaskB sets a field value that TaskA depends upon, the field value will show the correct value within the info window. The problem in this case is that TaskA assumes the field will be set before it is invoked. The two tasks are being evaluated out of order, which can be corrected.

If everything is **not** correct within the info windows, then you may also have a timing issue. The key difference is that you also have some additional clues to help you isolate the problem this time. In a situation like this, the erroneous field values and/or tags will point you at the specific areas where the problem resides. If a field has the wrong value, then whatever scripts are setting that value are basing their logic on information that hasn't been setup properly yet or that has been further modified. If the tags are incorrect, then whatever scripts are assigning/deleting the tags are basing their logic on similarly wrong information. Keying on the extra clues should make the process of isolating the problem significantly quicker and easier.

Category: Kit Reference

Establishing Timing Dependencies

Context: HL Kit … Kit Reference … Debugging Mechanisms

Contents

Overview

Timing issues are a major nuisance when you are developing your data files. They are often hard to spot when you first introduce them, which means that you can't always assume that the problem arose with the last few changes that you made. Timing issues typically entail a tedious diagnostic process that takes the fun out of creating new data files. Most importantly, though, timing issues can quickly degenerate into a game of "Whack-A-Mole".

All of the different tasks within your data files are like a big chain of dominoes. They all need to be evaluated in the correct sequence to ensure that you reach the final goal of having an accurate character. However, if you move one domino, then that has ripple effects on the dominoes that were before and after it. Consequently, if you change the timing of one task to resolve a timing issue, you could find yourself simply creating a **different** timing issue between other tasks that have inter-dependencies with the task you moved. Squish one timing bug and up pops another.

Managing a large number of tasks and keeping all the timing dependencies clearly understood and enforced can quickly become a nightmare for an aspiring data file author. So the Kit provides a convenient mechanism that allows you to instantly detect when a timing issue has been introduced. We call the mechanism "timing dependencies" and it allows you to setup dependencies between different tasks that the compiler will verify for you automatically. This mechanism is the focus of the topics below.

Naming Tasks

In order to establish a timing dependency on another task, you need to assign that task a name. By default, each task is assigned a name by the Kit, but it's not something you'll be able to establish a dependency upon. So any task that will be referenced within a dependency must be given a name. This is achieved by assigning the "name" attribute within the XML element that defines the task.

Task names can be just about anything you want. The goal is to identify what the task is doing clearly and succinctly, but you're the judge of how best to accomplish that. For example, you could name a task something detailed like "Calculate the Final Value for Attributes" or something brief like "Attr Final". The only criteria to keep in mind when choosing a name is that you'll need to type it anywhere that has a dependency.

Establishing "before" and "after" Dependencies

Named tasks become anchor points within the overall evaluation cycle for your data files. Other tasks will reference these anchors by establishing timing dependencies with them. This is accomplished by identifying the named task and specifying the nature of the dependency.

A dependency can be either a "before" or an "after" relationship. A "before" relationship tells the compiler that this task must always be performed before the named task. An "after" relationship indicates that the task must always be performed after the named task. Since all tasks are evaluated in a sequence, there is never a possibility that two tasks can be performed at the same time. Even if you assign them the identical phase and priority, the HL engine will assign an order to them. Consequently, each timing dependency must specify either a before or after relationship.

You can assign any number of timing dependencies to a given task. So you could define an assortment of dependencies that require a task to occur after certain tasks and an assortment that require that task to occur before others.

NOTE! Tasks do **not** need to be named in order to specify dependencies. Only tasks that serve as anchor points and are referenced by other tasks require names. Consequently, if you have a named task, you could have five different tasks depend on that task, and none of the dependent tasks require names.

Using the Timing Report

When your data files are compiled, a timing report is generated. You can view this timing report at any time by going to the "Debug" menu and selecting the "View Timing Report" option. For most users, your web browser will be launched and the file will be viewed within it. The timing report is an XML file that contains information about all of the timing dependencies you've defined and the tasks involved. Details on the contents will be found elsewhere within the Kit Reference documentation.

The most important sections in the timing report are the first three, so we'll discuss them here briefly. The first section identifies the names of any tasks that are unknown. These are names that are referenced by a timing dependency and that have not been defined. For example, if you specify that a task has a "before" dependency on the task named "My Task" and you have not assigned that name to any task, the name will be listed as unknown and no dependency will be established. You should always make sure that this section of the timing report is empty.

The second section identifies any names that have been duplicated and are not valid. It is perfectly legal to use the same name on multiple tasks. However, all tasks given the same name must be scheduled to occur at the exact same phase and priority. If you assign the name "My Task" to two separate tasks and they are scheduled at different times, they will be listed here as duplicates. You should always make sure that this section of the timing report is empty.

The third section identifies actual timing errors that exist between the tasks. If you specify that TaskA must occur before TaskB, and TaskA is assigned a phase and priority that is after TaskB, a timing error is reported and will be listed in this section. Any pair of tasks that does not satisfy the timing dependency assigned between them results in a timing error.

Compiler Checking of Dependencies

Whenever your data files are compiled, all of the timing dependencies are analyzed for you automatically. If the compiler identifies any errors in the timing dependencies you've specified, a corresponding error is reported. Unlike most errors, timing issues are treated merely as warning. This means that you are free to load the data files and use them, even though they will not work as you intended. The advantage of allowing you to load the files is that you can review both the timing report and the run-time information provided via info windows in an effort to resolve the problem.

With the timing dependency checks integrated into the compilation process, a critical safeguard is provided by HL. As you develop your files, you will add new logic and discover that the timing of some tasks needs to be adjusted in order to properly incorporate that new logic. If you change the timing of a task in order resolve a timing issue, you no longer have to worry about that change rippling into a different timing issue and having it go undetected. The compiler will use the timing dependencies to automatically detect whether the change has caused a ripple effect that also needs to be resolved. And the other sections of the timing report make it possible to pretty easily sort out even a chain of dependencies that become broken.

Deciding What Dependencies to Setup

After reading through all this, you might be thinking that you'll be spending a great deal of time setting up timing dependencies between all your tasks. That's not the case at all. In general, only a relatively small percentage of the tasks you define will have critical timing dependencies. Most tasks will be completely immune to dependency issues by simply assigning them to the appropriate phases. Consequently, you'll probably only find yourself needing to name maybe 10-20 tasks, and you'll likely only need to establish a similar number of dependencies upon those tasks.

The critical tasks that should be named as anchor points are those that satisfy three criteria. First, the task should be central to a series of evaluations that must occur in a specific order. Second, the tasks involved should be closely grouped within the evaluation cycle, such as all occurring with the same phase or a small number of sequential phases. Third, some of the tasks involved should have a reasonable chance of needing to be moved in the evaluation cycle as new logic is integrated into the data files.

If particular tasks satisfy all three of these criteria, then timing dependencies are extremely important. However, if only one or two of the criteria apply, then defining timing dependencies may or may not be justified. For example, if you have tasks that must setup values appropriately within fields before other tasks utilize those fields, setting up timing dependencies may be prudent. However, if the setup tasks are always performed within an early phase and the tasks using the values are always performed in a later phase, there is no need to establish timing dependencies.

Category: Kit Reference

Script Contexts

Context: HL Kit … Kit Reference

Contents

Overview

This first step in accessing data via scripts is in identifying where that data resides within the overall data hierarchy. A separate hierarchy is maintained for both structural information (e.g. actors, picks, gizmos, minions, etc.) and visual information (e.g. panels, layouts, templates, etc.). Each different layer within the hierarchy is considered a distinct "context".

IMPORTANT! Within certain script contexts, **all** target references are treated as read-only, regardless of whether they are normally writable. Any attempts to modify the contents of a target reference that has been forced to be read-only will fail to either compile or work at run-time. For example, if a Position script for a visual element attempts to modify the contents of a pick, it will be forbidden. Scripts and/or contexts within which this behavior occurs should specify it within their documentation.

Contexts Within Structural Hierarchy

Within the structural hierarchy, there are a variety of contexts managed by the Kit. The following table identifies these contexts and provides a brief description of what each represents.

portfolio	The "portfolio" context represents the topmost level within the structural hierarchy, encapsulating all of the different leads that have been created. This context is generally **not** accessible via scripts, as each lead is considered to be an atomic object.
hero	The "hero" context represents an individual actor within the portfolio. This actor could be a lead or a minion.
container	The "container" context represents any container within the portfolio. The container could be an actor or a gizmo.
pick	The "pick" context represents any pick throughout the portfolio, located within any container.
thing	The "thing" context represents a thing that has not been added to the portfolio. The thing context is very similar to the pick context in behavior, with the only real difference being that the dynamic facets of picks don't exist for things, resulting in many valid actions for picks being inaccessible from things.
field	The "field" context represents any field within any pick or thing. If within a thing, all aspects of the field are read-only.
pool	The "pool" context represents any usage pool associated with either the actor or a specific pick.

There are also quite a few logical contexts within the structural hierarchy. Each of these contexts actually maps to one of the basic contexts above, but it is identified below due to important considerations, such as added restrictions or limitations on what can be done within the context.

parent	The "parent" context is wholly dependent on the current context. If the current context is a pick, then the parent is the container that the pick resides within. If the current context is a container, then the parent is the pick that attaches the gizmo, unless the container is an actor, in which case there is no parent.
linkage	The "linkage" context proceeds from one pick to another pick that has been associated with the first pick via a named linkage. If no linkage has been established, then there is no linkage context.
root	The "root" context identifies the pick that bootstrapped the current pick. If the pick was not bootstrapped by another pick, then there is no root context.
dynalink	The "dynalink" context provides access from one pick to another pick that is dynamically setup via the Creation script for the original pick. If no dynamic link is setup for a pick, there is no dynalink context.
gearholder	The "gearholder" context accesses the pick that is designated as the "holder" of the current pick. If the pick is not gear or is not presently assigned to another piece of gear as a holder, there is no gearholder context.
shadow	The "shadow" context represents the shadowed version of a pick, which resides within a different location in the structural hierarchy due the shadowing. If the pick is not shadowed, there is no shadow context.
origin	The "origin" context represents the original version of a displaced pick, which resides within a different location in

	the structural hierarchy due to the displacement. If the pick is not displaced, there is no origin context.
master	The "master" context identifies the actor that is the master of the current actor. If the current actor is not a minion, there is no master context.
minion	The "minion" context identifies an actor that is a minion of the current actor. If the current actor is not a master, there is no minion context.
anchor	The "anchor" context identifies the specific pick that attaches the current actor as a minion. If the current actor is not a minion, there is no anchor context.
child	The "child" context accesses a specific child pick that exists within the current container.
gizmo	The "gizmo" context references the gizmo that is attached as a child of the current pick. If the pick does not attach a gizmo, there is no gizmo context.

Contexts Within Visual Hierarchy

The visual hierarchy has a separate set of contexts that are managed by the Kit and that are accessed exclusively via Position scripts on visual elements. The following table describes each of these contexts.

scene	The "scene" context identifies the top-level visual element within the current hierarchy, which will be either a panel, a form, or a sheet.
layout	The "layout" context identifies a layout within the top-level scene of the current hierarchy.
template	The "template" context identifies a template within the current hierarchy. Templates can either be used within layouts or within tables, so the parent context of the template within the hierarchy can vary.
portal	The "portal" context identifies a portal within the current hierarchy. Portals can either be used within templates or within layouts, so the parent context of the portal within the hierarchy can vary.
table	The "table" context identifies the containing table of a template or portal within the current hierarchy.
value	The "value" context represents any field within the pick or thing associated with a template. The value context is used for display only, so all aspects of the field are always read-only.

There are also a few logical contexts within the visual hierarchy. Each of these contexts actually maps to a basic context, but it is identified below due to important considerations, such as added restrictions or limitations on what can be done within the context.

parent	The "parent" context is wholly dependent on the current context, as detailed below. ▪ portal – The parent is the visual container that the portal resides within (i.e. a template or a layout). ▪ template – The parent is the visual container that the template resides within (i.e. a layout or a table). ▪ layout – The parent is the containing scene. ▪ scene – There is no parent for a scene.
chosen	The "chosen" context represents the current selection within a thing-based menu, so it can either be a pick or thing.
hero	The "hero" context within a visual script identifies the actor that the visual element is operating upon. This context is always treated as read-only from within a visual script.
container	The "container" context within a visual script identifies the container that the visual element is operating upon. This context is always treated as read-only from within a visual script.
value	The "value" context within a visual script represents the contents of a field. The field is accessed via a pick, which is identified by the template from which the value context is being utilized.

General Contexts

In addition to the structural and visual contexts, the Kit supports contexts that are outside of the the normal hierarchy. These general contexts are described in the table below.

state	The "state" context provides access to overall state information that pertains to the portfolio as a whole or general information about the evaluation cycle.
transaction	The "transaction" context represents the special pick that is utilized for buy and sell transactions. Since such transactions can be canceled by the user, all the details must be managed in a temporary fashion until the user

	completes the transaction.
focus	The "focus" context reflects the pick that has been established as the current "pick focus" via explicit actions within scripts. If no pick focus has been setup, then there is no focus context.
actor	The "actor" context reflects the hero that has been established as the current "actor focus" via explicit actions within scripts. If no actor focus has been setup, then there is no actor context.
eachpick	The "eachpick" context reflects the current pick that is being processed within a "foreach" loop performed by a script.
altpick	The "altpick" context identifies a secondary pick that is involved in an operation, such as when merging two stackable picks into one.
altthing	The "altthing" context identifies a secondary thing that is involved in an operation, such as is necessary when performing pre-requisite tests.

Category: Kit Reference

Hero Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "hero" context represents an individual actor within the portfolio. This actor could be a lead or a minion.

Context Transitions

A "hero" context is very similar to a "container" context, except that the options available are a bit more limited (e.g. heroes don't have parents). From within a "hero" context, you can utilize the following set of valid context transitions:

state	Transitions to the state context. Example: this.state
child[*id*]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the *id* specified. If the container has no child pick with the given unique id, a run-time error notification is reported to the user and the transition fails to resolve. Example: this.child[mypick]
childfound[*id*]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the *id* specified. This transition is identical to "child[id]", except that the existence of the child pick is optional. If the child is found, the transition occurs normally. If the child does not exist, no run-time error is reported, although the transition still fails to resolve. Example: this.childfound[mypick]
firstchild[*expr*,*sort*]	Transitions to the pick context corresponding to the first pick within the container that satisfies the tag expression given by *expr*. Since multiple children may satisfy the tag expression, an optional sort set id may be specified by *sort*, resulting in all matching children being sorted and the first child being used after the sort is performed. The tag expression may be either a literal string or a string expression. If no matching child exists, the transition fails to resolve. Example: this.firstchild[expr,mysort] Example: this.firstchild[expr]
minion[*id*]	Transitions to the **hero context** corresponding to the minion actor with the given *id* that exists beneath the actor that possesses the current container. If the container is not a master or the specified minion does not exist, the transition fails to resolve. Example: this.minion[myminion]
herofield[*id*]	Transitions to the field context corresponding to the field given by *id* that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]". Example: this.herofield[myfield]
usagepool[*id*]	Transitions to the pool context corresponding to the usage pool given by *id* that exists within the current actor. Example: this.usagepool[mypool]
transact	Transitions to the pick context corresponding to the transaction pick that is associated with the hero governing the current context. Example: this.transact **NOTE!** The transaction pick is only utilized within buy and sell transactions. As such, this transition is only valid within a few select scripts.
dynalink[*index*]	Transitions to the pick context corresponding to the registered dynamic linkage with the *index* specified. If no dynamic linkage has been registered with the given *index*, the transition fails to resolve. The *index* may be an arithmetic expression that calculates the actual index value to be used. Example: this.dynalink[myindex]

Target References

Heroes are a special type of container. As such, they share all of the same target references as normal containers. However, they also have quite a few additional target references that are unique to heroes. The complete list of these special target references is presented in the table below.

IMPORTANT! For the purposes of data file authoring, the "hero" context applies to all actors, whether they be leads, masters, or minions.

IMPORTANT! Actors also support all general container target references.

NOTE! When an actor is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a container from within a visual script will fail.

setactor	(Right, Number) Memorizes the current actor context as the "actor focus", allowing it to be instantly accessed thereafter via the "actor." initial script context. Always returns a value of zero. Example: perform this.setactor
miniontext	(Right, String) Returns the name of the thing that attaches the current minion. This makes it possible to retrieve the nature of the minion for display to the user. If invoked on a lead, an empty string is returned. Example: result = this.miniontext
sourcetree	(Right, String) Returns a summary of the various user-selected sources that have chosen for the current actor. This is intended for inclusion within character sheet output. The summary is synthesized solely from sources that are user-selected, although the entire chain of sources down to each selected source is included so that context is provided in case similar names are used in different contexts. Any source that is designated as not reportable is omitted from the summary, allowing sources governing printout and interface behaviors to be omitted. If a source specifies an alternate "reportname", that name is used instead. Example: result = this.sourcetree
errorcount	(Right, Number) Returns the total number of validation errors that were reported within the hero over the course of evaluation. This is intended for use within character sheet output. If accessed during evaluation processing, a run-time error is reported. Example: result = this.errorcount
errorlist	(Right, String) Returns a string containing all validation errors for the hero, based on the most recent evaluation cycle. This is intended for use within character sheet output. If accessed during evaluation processing, a run-time error is reported. Example: result = this.errorlist
combatant	(Right, Left, Number) Indicates whether the actor is managed as a combatant or not within the Tactical Console, with a non-zero value indicating a combatant. If a new value is assigned, the combatant state of the actor is changed. Example: result = this.combatant Example: this.combatant = 0
initchange	(Right, Number) Returns non-zero if the actor's current initiative value has been modified by the user. Example: result = this.initchange
ismoveup	(Right, Number) Returns non-zero if the actor can be moved upwards within the Tactical Console, thereby adjusting its initiative value. Example: result = this.ismoveup
ismovedown	(Right, Number) Returns non-zero if the actor can be moved downwards within the Tactical Console, thereby adjusting its initiative value. Example: result = this.ismovedown
combatmove[*expr*]	(Right, Number) Moves the actor upwards or downwards within the Tactical Console, thereby adjusting its initiative position. The number of slots to move is given by the numeric expression *expr*, with a positive value moving the actor later in the initiative sequence and a negative value earlier. To move an actor down in the order (i.e. later in combat), use "+1", and use "-1" to move the actor upwards. To move an actor to the top or bottom of the initiative order, simply use a very large value, as the engine will safely bound all adjustments. Always returns a value of zero. Example: perform this.combatmove[-1]
combatact	(Right, Number) Designates the actor as having acted this combat turn and triggers an update to determine the next "ready" actor(s). Always returns a value of zero. Example: perform this.combatact
combatdefer	(Right, Number) Designates the actor as having deferred its action this combat turn and triggers an update to determine the next "ready" actor(s). Always returns a value of zero.

	Example: perform this.combatdefer
combatmovenow	(Right, Number) Moves the actor within the initiative sequence such that it is positioned immediately **before** the first actor that is currently designated as "ready". This is intended for use when a deferred actor finally acts, as its initiative position is moved to the current slot in the queue. Always returns a value of zero. Example: perform this.combatmovenow

Category: Kit Reference

Container Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "container" context represents any container within the portfolio. The container could be an actor or a gizmo.

IMPORTANT! If the container context happens to be an actor, then you can utilize the context as either a standard container context **or** as a Hero Context.

Context Transitions

From within a "container" context, you can utilize the following set of valid context transitions:

state	Transitions to the state context. Example: this.state
hero	Transitions to the hero context corresponding to the hero that is the parent of the current container. If the current container **is** a hero, then this transition changes nothing but does resolve successfully. Example: this.hero
container	Transitions to the **container context** corresponding to whatever container is the immediate parent of the current container. If the current container is a hero, then the transition fails to resolve. Example: this.container
parent	Transitions to the pick context corresponding to the parent pick that attaches the container. If the container is a hero and has no parent pick, the transition fails to resolve. Example: this.parent
child[*id*]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the *id* specified. If the container has no child pick with the given unique id, a run-time error notification is reported to the user and the transition fails to resolve. Example: this.child[mypick]
childfound[*id*]	Transitions to the pick context corresponding to the first pick within the container that derives from the thing with the *id* specified. This transition is identical to "child[id]", except that the existence of the child pick is optional. If the child is found, the transition occurs normally. If the child does not exist, no run-time error is reported, although the transition still fails to resolve. Example: this.childfound[mypick]
firstchild[*expr*,*sort*]	Transitions to the pick context corresponding to the first pick within the container that satisfies the tag expression given by *expr*. Since multiple children may satisfy the tag expression, an optional sort set id may be specified by *sort*, resulting in all matching children being sorted and the first child being used after the sort is performed. The tag expression may be either a literal string or a string expression. If no matching child exists, the transition fails to resolve. Example: this.firstchild[expr,mysort] Example: this.firstchild[expr]
anchor	Transitions to the pick context corresponding to the pick within the master actor that attaches the current actor as a minion. If the container does not reside within a minion, the transition fails to resolve. Example: this.anchor
master	Transitions to the hero context corresponding to the master actor for which this container is a minion. If the container is not a minion, the transition fails to resolve. Example: this.master
minion[*id*]	Transitions to the hero context corresponding to the minion actor with the given *id* that exists beneath the actor that possesses the current container. If the container is not a master or the specified minion does not exist, the transition fails to resolve. Example: this.minion[myminion]
herofield[*id*]	Transitions to the field context corresponding to the field given by *id* that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]". Example: this.herofield[myfield]
	Transitions to the pool context corresponding to the usage pool given by *id* that exists within the current

usagepool[*id*]	container. This transition is only valid for actors, since gizmos do not possess usage pools. Example: this.usagepool[mypool]
transact	Transitions to the pick context corresponding to the transaction pick that is associated with the hero governing the current context. Example: this.transact **NOTE!** The transaction pick is only utilized within buy and sell transactions. As such, this transition is only valid within a few select scripts.
dynalink[*index*]	Transitions to the pick context corresponding to the registered dynamic linkage with the *index* specified. If no dynamic linkage has been registered with the given *index*, the transition fails to resolve. The *index* may be an arithmetic expression that calculates the actual index value to be used. Example: this.dynalink[myindex]

Target References

There are a wide assortment of operations that can be performed on containers, some of which modify the container and some that simply retrieve information about the container. The container context applies equally to heroes and gizmos, although there are some behavioral differences that arise for some target references. The complete list of target references for containers is presented in the table below.

NOTE! When a container is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a container from within a visual script will fail.

live	(Right, Number) Returns non-zero if the container is currently considered "live", else zero if non-live. The live state for a gizmo is dictated by the live state of its parent pick that attaches it, while a hero is always considered live. Example: result = this.live
ishero	(Right, Number) Returns non-zero if the container is a hero, else zero for a gizmo. Example: result = this.ishero
isactive	(Right, Number) Returns non-zero if the container either is or resides within the currently active actor within the HL interface. If the container is or resides within a different actor, zero is returned. Example: result = this.isactive
livename	(Right, String) Returns a suitable name for the container that is based on dynamic changes made via scripts. If the container is an actor, the name of the actor is returned, else the name of the parent pick of the gizmo is returned. Example: result = this.livename
actorname	(Right, String) Returns the name of the actor that encompasses the current container context. The name is whatever has been assigned by the user. Example: result = this.actorname
playername	(Right, String) Returns the name of the player that is associated with the current lead. The name is whatever has been entered by the user. Example: result = this.playername
assign[*tag*]	(Right, Number) Assigns the indicated *tag* to the container. The tag must be specified using the standard "group.id" syntax. The tag is verified to exist during compilation of the script. Always returns a value of zero. Example: perform this.assign[skill.appraise]
delete[*tmpl*]	(Right, Number) Deletes all tags from the container that match the tag template *tmpl*. The template must use the standard "group.id" syntax and may contain a wildcard. If the template employs a wildcard, all tags matching the template are deleted. If the template specifies an explicit tag, and the tag has been assigned to the container multiple times, the tag is deleted only once, thereby providing detailed control to authors when needed. Always returns a value of zero. Example: perform this.delete[skill.craft?]
	(Right, Number) This target reference is identical to "assign", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag to be dynamically

assignstr[*str*]	determined via the script instead of being hard-wired at compilation. Always returns a value of zero. Example: perform this.assignstr[skill.appraise]
deletestr[*str*]	(Right, Number) This target reference is identical to "delete", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero. Example: perform this.deletestr["skill.craft?"]
tagis[*tmpl*]	(Right, Number) Returns non-zero if any tags assigned to the container match the tag template *tmpl*, else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard. Example: this.tagis[skill.?]
tagcount[*tmpl*]	(Right, Number) Returns the number of tags assigned to the container that match the tag template *tmpl*. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagcount[skill.?]
tagvalue[*tmpl*]	(Right, Number) Returns the value of a tag assigned to the container that matches the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagvalue[spelllevel.wizard?]
tagmin[*tmpl*]	(Right, Number) Returns the minimum value of all tags assigned to the container that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmin[spelllevel.wizard?]
tagmax[*tmpl*]	(Right, Number) Returns the maximum value of all tags assigned to the container that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmax[spelllevel.wizard?]
tagunique[*tmpl*]	(Right, Number) Returns the number of unique tags assigned to the container that match the tag template *tmpl*. If a tag is assigned multiple times, it is only counted once. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagunique[skill.?]
tagexpr[*expr*]	(Right, Number) Returns non-zero if the tags assigned to the container match the tag expression *expr*, else zero is returned. The *expr* parameter must be a valid tag expression. Example: result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]
tagcountstr[*str*]	(Right, Number) This target reference is identical to "tagcount", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagcountstr["skill.?"]
tagvaluestr[*str*]	(Right, Number) This target reference is identical to "tagvalue", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagvaluestr["spelllevel.wizard?"]
tagminstr[*str*]	(Right, Number) This target reference is identical to "tagmin", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagminstr["spelllevel.wizard?"]
tagmaxstr[*str*]	(Right, Number) This target reference is identical to "tagmax", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagmaxstr["spelllevel.wizard?"]
	(Right, Number) This target reference is identical to "tagunique", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be

taguniquestr[*str*]	dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.taguniquestr["skill.?"]
tagsearch[*str*]	(Right, Number) This target reference is identical to "tagexpr", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"] **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[*refgrp*,*match*,*ref*]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here. Example: result = hero.tagmatch[NeedStatus,HasStatus,initial]
heromatch[*refgrp*,*match*,*ref*]	(Right, Number) Returns non-zero if any tags within the actor also exist within a match context, else zero. This allows tags from the actor to be verified to exist within another context, making tag-based comparisons between the actor and another object possible. The key difference between "heromatch" and "tagmatch" is that this target reference always assumes the initial context is the actor, which is then compared against the context that is transitioned to. For more details, please check here. Example: result = this.heromatch[NeedStatus,HasStatus,initial]
intersect[*init*,*curr*]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the currently transitioned context, else zero. All of the tags that belong to the *init* tag group within the initial script context are compared against the tags that belong to the *curr* tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing). Example: result = this.interset[MyGroup,AltGroup]
inherit[*id*]	(Right, Number) The container inherits all tags from one or all of its child picks. If an *id* is given, then all tags from the pick with that id are inherited into the container. If no parameter is given, then all tags from all child picks are inherited. The return value is the total number of tags inherited. Example: result = this.inherit[mypick] Example: result = this.inherit
pulltags[*tmpl*,*grp*]	(Right, Number) Copies tags from the transitioned container context into the initial pick or container context and returns the total number of tags copied. The set of tags to be copied from the transitioned context are dictated by the tag template *tmpl*. If the second parameter is omitted, the identified tags are copied to the initial context. If the second parameter is provided, *grp* must specify a tag group. All tags identified by *tmpl* are mapped to equivalent tags within the tag group *grp*, and those mapped tags are then assigned to the initial context. When mapping, an equivalent mapped tag must exist for all identified tags. Both the initial and transitioned script context must be either picks or containers. Example: result = hero.pulltags[thingid.?] Example: result = hero.pulltags[thingid.?,Ability]
pushtags[*tmpl*,*grp*]	(Right, Number) Copies tags from the initial pick or container context into the transitioned container context and returns the total number of tags copied. This target reference is identical in behavior to "pulltags", except that the tags are copied in the opposite direction. Example: result = hero.pushtags[thingid.?] Example: result = hero.pushtags[thingid.?,Ability]
childexists[*id*]	(Right, Number) Returns non-zero if any child pick with the given *id* exists within the container, else zero if no matching pick is found. Example: result = this.childexists[mypick]
childlives[*id*]	(Right, Number) Returns non-zero if any child pick with the given *id* exists within the container **and** is "live". If either no matching pick is found or all matching picks found are "non-live", zero is returned. Example: result = this.childlives[mypick]

childcount[*id*]	(Right, Number) Returns the number of child picks with the given *id* that exist within the container. A value of zero indicates that no matching picks were found. Example: result = this.childcount[mypick]
haschild[*str*]	(Right, Number) Returns the number of child picks within the container that match the tag expression given by *str*. The parameter is a string expression that must contain a valid tag expression and is tested against all children of the container. The parameter can be synthesized dynamically within the script. Example: result = this.haschild["component.Skill"]
setidentity[*grp*]	(Right, Number) Assigns to the container the identity tag from the tag group *grp* that corresponds to the initial context of the script. The identity tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.setidentity[groupid]
isidentity[*grp*]	(Right, Number) As the counterpart of "setidentity", this target reference returns non-zero if the indicated identity tag has been assigned to the container and zero if not. The identity tag sought must be from the tag group *grp* and the tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.isidentity[groupid]
countidentity[*grp*]	(Right, Number) Similar to "isidentity", this target reference returns the count of the identity tags assigned to the container. The identity tag sought must be from the tag group *grp* and the tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.countidentity[groupid]
isminion	(Right, Number) Returns non-zero if the container is or resides within a minion, else zero. Example: result = this.isminion
ismaster	(Right, Number) Returns non-zero if the container is or resides within a master, else zero. Example: result = this.ismaster
hasminion[*id*]	(Right, Number) Returns non-zero if the container is or resides within an actor that possesses a minion with the specified *id*. If the container is not within a master or does not contain the specified minion, zero is returned. Example: result = this.hasminion[myminion]
isdynalink[expr]	(Right, Number) Returns non-zero if a dynamic linkage has been defined for the implied hero context with the index specified, else zero if no linkage exists. If the container is a gizmo, then the containing actor is used as the hero context. The value given by index may be an arithmetic expression that will be resolved properly at run-time. Example: result = this.isdynalink[4]
istransact	(Right, Number) Returns non-zero if a viable transaction context exists for the container. This allows scripts to check that a transaction context exists before attempting to transition into the transaction context. Example: result = this.istransact
panelvalid[*id*]	(Left, Number) Sets the validity state of the tab panel given by *id*. If the value assigned is zero, the panel is designated as non-valid and its name will be highlighted in red to the user. Since the default state of all panels is valid, if a non-zero value is assigned, this target reference is ignored. This means you can simply designate a panel as invalid when appropriate and do nothing when the panel is valid. Example: result = this.panelvalid[mypanel] **NOTE!** This target reference can only be used from within an Eval Rule or a Validate script.
tagnames[*tmpl*,*spl*]	(Right, String) Generates and returns a list of tags within the container. Only tags that match the tag template *tmpl* are included in the list. The generated string appends the names of the tags together, inserting the splicing string *spl* between each name if there is more than one. A container with no tags matching the template will return the empty string. Example: result = this.tagnames[Weapon.?,"+"]
tagabbrevs[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names. Example: result = this.tagabbrevs[Weapon.?,"+"]

tagids[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the unique ids for all matching tags instead of their names. Example: result = this.tagids[Weapon.?,"+"]
weight	(Right, Number) Returns the total weight of all "gear" picks within the container. Example: result = this.weight
gearlist[*spl*,*expr*]	(Right, String) Generates and returns a list of gear held within the container, which means gear that is **not** held within another gear holder. Only picks that are designated as gear and that are not assigned to a holder are candidates for inclusion in the list. Each candidate piece of gear is compared against the tag expression *expr*, and only those that satisfy the tag expression are included in the final list. The generated string appends the names of all pieces of gear together, inserting the splicing string *spl* between each name if there is more than one. A container that holds no gear matching the tag expression will return the empty string. Example: result = this.gearlist["+",!Equipment.Natural]
geartree[*expr*]	(Right, String) Generates and returns a hierarchical tree view of the gear picks possessed by the container context, as appropriate for use within the Dashboard and Tactical Console. Only picks that are designated as gear are candidates for inclusion in the report. Each candidate piece of gear is compared against the tag expression *expr*, and only those that satisfy the tag expression are included in the final report. All gear is output in a hierarchy, where each level of containment is indented beneath the level above it. Gear within child gizmos is also included, with indentation as if the gizmo were a holder within the container and its contents indented beneath it. Example: result = this.geartree[!Helper.NoMove]

Category: Kit Reference

Pick Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "pick" context represents any pick throughout the portfolio, located within any container.

Context Transitions

From within a "pick" context, you can utilize the following set of valid context transitions:

state	Transitions to the state context. Example: this.state
hero	Transitions to the hero context corresponding to the hero that contains the current pick. Example: this.hero
container	Transitions to the container context corresponding to the immediate container of the current pick, whether it be a hero or a gizmo. Example: this.container
parent	Transitions to the container context corresponding to the immediate container of the current pick, just like the "container" transition. Example: this.parent
gizmo	Transitions to the container context corresponding to the child gizmo directly attached by the pick. If the pick has no attached child gizmo, the transition fails to resolve. Example: this.gizmo
field[*id*]	Transitions to the field context corresponding to the field within the current pick that has the *id* specified. If the given field does not exist for the current pick, the transition fails to resolve. Example: this.field[myfield] **NOTE!** Within things and picks, there are a number of pre-defined pseudo-fields that are always defined and that allow access to facets of the pick that are not governed by user-defined fields. These pseudo-fields behave like normal fields in all respects within scripts, except that some are read-only. The list of pre-defined fields can be found elsewhere in the Kit Reference documentation.
root	Transitions to the **pick context** corresponding to the root pick that bootstraps the current pick into the container. If the current pick is not bootstrapped, or if the current pick is designated as unique and can possess multiple bootstraps, the transition fails to resolve. Example: this.source
gearholder	Transitions to the **pick context** corresponding to the pick that is assigned as the gear holder of the current pick. If the current pick is not held, the transition fails to resolve. If the current pick is not gear, an error is reported and the transition fails to resolve. Example: this.gearholder
linkage[*id*]	Transitions to the **pick context** corresponding to the linkage with the *id* specified. If the linkage is not defined, an error is reported. Example: this.linkage[mylink]
anchor	Transitions to the **pick context** corresponding to the pick within the master actor that attaches the current actor as a minion. If the pick does not reside within a minion, the transition fails to resolve. Example: this.anchor
master	Transitions to the hero context corresponding to the master actor for which this pick's container is a minion. If the pick is not within a minion, the transition fails to resolve. Example: this.master
minion[*id*]	Transitions to the hero context corresponding to the minion actor with the given *id* that exists beneath the actor that possesses the current pick. The *id* can be omitted, in which case the minion is implicitly identified and must be directly attached by the current pick. If the pick does not reside within a master or the specified minion does not exist, the transition fails to resolve. Example: this.minion[myminion] Example: this.minion

herofield[*id*]	Transitions to the field context corresponding to the field given by *id* that exists on the "actor" pick for the containing actor. This is a shorthand notation for "hero.child[actor].field[id]". Example: this.herofield[myfield]
usagepool[*id*]	Transitions to the pool context corresponding to the usage pool with the *id* specified for the current pick. Example: this.usagepool[mypool]
shadow	Transitions to the container context corresponding to the container into which the current pick is shadowed. If the pick is not shadowed, an error is reported and the transition fails to resolve. Example: this.shadow
origin	Transitions to the container context corresponding to the container into which the current pick was originally added. If the pick is displaced, the container is where the pick was added by the user. If not displaced, the container is simply the container for the pick. Example: this.origin

Target References

Picks are going to be the most prevalent object type within any set of data files, so it's no surprise that picks have the largest and most varied assortment of target references. The complete list of target references for picks is presented in the table below.

NOTE! When a pick is accessed from within a visual script, all operations are restricted to read-only behavior. Any attempts to modify a pick from within a visual script will fail.

livename	(Right, String) Returns an appropriate name for the pick. If the user has explicitly named the pick, that name is returned. If not named by the user, any name change dictated by scripts is returned. Otherwise, the name of the thing is used. Example: result = this.livename
idstring	(Right, String) Returns the unique id of the thing as a string. This can be extremely handy when synthesizing tag templates and tag expressions on-the-fly via scripts. Example: result = this.idstring
valid	(Right, Number) Returns non-zero if the pick is valid and zero if the pick has been designated as non-valid. Validity is controlled via mechanisms like pre-requisite tests and Eval Rules. Example: result = this.valid
isuser	(Right, Number) Returns non-zero if the pick was directly added by the user, else zero. Picks that are bootstrapped by containers or other picks are not considered user-added, even if the pick that does the bootstrapping **is** user-added. Example: result = this.isuser
ispick	(Right, Number) Returns non-zero if the current context is a pick and zero if it's a thing. This provides a means to discern the nature of the context when the circumstances make a distinction uncertain, such as within procedures. Example: result = this.ispick
isroot	(Right, Number) Returns non-zero if the pick has been bootstrapped and therefore has a root pick available. Example: result = this.isroot
isgizmo	(Right, Number) Returns non-zero if the pick directly attaches a child gizmo. Example: result = this.isgizmo
isentity	(Right, Number) Returns non-zero if the pick directly attaches a child entity. Example: result = this.isentity **NOTE!** This target reference is essentially identical to "isgizmo" (above).
isunique	(Right, Number) Returns non-zero if the pick behaves as unique. Example: result = this.isunique
shadowed	(Right, Number) Returns non-zero if the pick has been shadowed and also exists within an alternate container. Example: result = this.shadowed

displaced	(Right, Number) Returns non-zero if the pick has been displaced and also exists within an alternate container. Example: result = this.displaced
countme	(Right, Number) Returns the total number of instances of the current pick that exist within the container of the current pick. As long as the thing id matches, two picks are considered equivalent. If stacking is utilized, this target reference returns the total number of distinct picks within the container, ignoring all stacked quantities. Example: result = this.countme
uniqcount	(Right, Number) Return the number of times that a unique pick has been added to the current container. Every time a unique pick is added, whether by the user or via bootstrapping, it's reference count is incremented, and this target reference returns the reference count. If the pick is not unique, a run-time error is reported and zero is returned. Example: result = this.uniqcount
assign[*tag*]	(Right, Number) Assigns the indicated *tag* to the pick. The tag must be specified using the standard "group.id" syntax. The tag is verified to exist during compilation of the script. Always returns a value of zero. Example: perform this.assign[skill.appraise]
delete[*tmpl*]	(Right, Number) Deletes all tags from the pick that match the tag template *tmpl*. The template must use the standard "group.id" syntax and may contain a wildcard. If the template employs a wildcard, all tags matching the template are deleted. If the template specifies an explicit tag, and the tag has been assigned to the pick multiple times, the tag is deleted only once, thereby providing detailed control to authors when needed. Always returns a value of zero. Example: perform this.delete[skill.craft?]
assignstr[*str*]	(Right, Number) This target reference is identical to "assign", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero. Example: perform this.assignstr[skill.appraise]
deletestr[*str*]	(Right, Number) This target reference is identical to "delete", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Always returns a value of zero. Example: perform this.deletestr["skill.craft?"]
tagis[*tmpl*]	(Right, Number) Returns non-zero if any tags assigned to the pick match the tag template *tmpl*, else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard. Example: this.tagis[skill.?]
tagcount[*tmpl*]	(Right, Number) Returns the number of tags assigned to the pick that match the tag template *tmpl*. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagcount[skill.?]
tagvalue[*tmpl*]	(Right, Number) Returns the value of a tag assigned to the pick that matches the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagvalue[spelllevel.wizard?]
tagmin[*tmpl*]	(Right, Number) Returns the minimum value of all tags assigned to the pick that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmin[spelllevel.wizard?]
tagmax[*tmpl*]	(Right, Number) Returns the maximum value of all tags assigned to the pick that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmax[spelllevel.wizard?]
	(Right, Number) Returns the number of unique tags assigned to the pick that match the tag template *tmpl*. If a tag is assigned multiple times, it is only counted once. The template must

tagunique[*tmpl*]	use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagunique[skill.?]
tagexpr[*expr*]	(Right, Number) Returns non-zero if the tags assigned to the pick match the tag expression *expr*, else zero is returned. The *expr* parameter must be a valid tag expression. Example: result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]
tagcountstr[*str*]	(Right, Number) This target reference is identical to "tagcount", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagcountstr["skill.?"]
tagvaluestr[*str*]	(Right, Number) This target reference is identical to "tagvalue", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagvaluestr["spelllevel.wizard?"]
tagminstr[*str*]	(Right, Number) This target reference is identical to "tagmin", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagminstr["spelllevel.wizard?"]
tagmaxstr[*str*]	(Right, Number) This target reference is identical to "tagmax", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagmaxstr["spelllevel.wizard?"]
taguniquestr[*str*]	(Right, Number) This target reference is identical to "tagunique", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.taguniquestr["skill.?"]
tagsearch[*str*]	(Right, Number) This target reference is identical to "tagexpr", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"] **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[*refgrp*,*match*,*ref*]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here. Example: result = hero.tagmatch[NeedStatus,HasStatus,initial]
heromatch[*refgrp*,*match*,*ref*]	(Right, Number) Returns non-zero if any tags within the actor also exist within a match context, else zero. This allows tags from the actor to be verified to exist within another context, making tag-based comparisons between the actor and another object possible. The key difference between "heromatch" and "tagmatch" is that this target reference always assumes the initial context is the actor, which is then compared against the context that is transitioned to. For more details, please check here. Example: result = hero.heromatch[NeedStatus,HasStatus,initial]
intersect[*init*,*curr*]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the currently transitioned context, else zero. All of the tags that belong to the *init* tag group within the initial script context are compared against the tags that belong to the *curr* tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing). Example: result = this.interset[MyGroup,AltGroup]
	(Right, Number) Copies tags from the transitioned pick context into the initial pick or container context and returns the total number of tags copied. The set of tags to be copied from the transitioned context are dictated by the tag template *tmpl*. If the second parameter is omitted,

pulltags[*tmpl*,*grp*]	the identified tags are copied to the initial context. If the second parameter is provided, *grp* must specify a tag group. All tags identified by *tmpl* are mapped to equivalent tags within the tag group *grp*, and those mapped tags are then assigned to the initial context. When mapping, an equivalent mapped tag must exist for all identified tags. Both the initial and transitioned script context must be either picks or containers. Example: result = this.pulltags[thingid.?] Example: result = this.pulltags[thingid.?,Ability]
pushtags[*tmpl*,*grp*]	(Right, Number) Copies tags from the initial pick or container context into the transitioned pick context and returns the total number of tags copied. This target reference is identical in behavior to "pulltags", except that the tags are copied in the opposite direction. Example: result = this.pushtags[thingid.?] Example: result = this.pushtags[thingid.?,Ability]
forward[*tmpl*,*target*]	(Right, Number) Copies tags from the pick context into the corresponding container and returns the total number of tags copied. The set of tags to be copied are dictated by the tag template *tmpl*. The tags are copied to the container dictated by *target*, which must be one of the following values: ▪ parent – Tags are copied to the standard parent container of the pick. ▪ shadow – Tags are copied to the shadow container of the pick, which is only applicable when the pick has been shadowed. ▪ origin – Tags are copied to the origin container of the pick, which is only applicable when the pick has been displaced. The *target* parameter may be omitted, in which case all tags matching the tag template are copied to the "parent" container. Both parameters may also be omitted, in which case all tags within the pick are copied to the "parent" container. Example: result = this.forward[thingid.?,shadow] Example: result = this.forward[thingid.?] Example: result = this.forward
setidentity[*grp*]	(Right, Number) Assigns to the pick the identity tag from the tag group *grp* that corresponds to the initial context of the script. The identity tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.setidentity[groupid]
isidentity[*grp*]	(Right, Number) As the counterpart of "setidentity", this target reference returns non-zero if the indicated identity tag has been assigned to the pick and zero if not. The identity tag sought must be from the tag group *grp* and the tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.isidentity[groupid]
countidentity[*grp*]	(Right, Number) Similar to "isidentity", this target reference returns the count of the identity tags assigned to the pick. The identity tag sought must be from the tag group *grp* and the tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.countidentity[groupid]
shareidentity[*grp*,*pick*]	(Right, Number) The pick with unique id *pick* is assigned the identity tag from the tag group *grp* for the current pick context. This allows a script to explicitly identify a pick and assign an identity tag to it. The identity tag id is dictated by the initial context of the script. For more details, please check here. If the specified pick does not exist, a run-time error is reported. A value of zero is always returned. Example: perform this.shareidentity[ClassSkill,mypick]
pullidentity[*grp*]	(Right, Number) Locates the identity tag from the tag group *grp* for the current pick context and assigns it to the initial script context. The identity tag id is dictated by the initial context of the script. For more details, please check here. If the initial script context is neither a pick nor a container, a run-time error is reported. A value of zero is always returned. Example: perform this.pullidentity[groupid]
tagnames[*tmpl*,*spl*]	(Right, String) Generates and returns a list of tags within the pick. Only tags that match the tag template *tmpl* are included in the list. The generated string appends the names of the tags together, inserting the splicing string *spl* between each name if there is more than one. A pick

	with no tags matching the template will return the empty string. Example: result = this.tagnames[Weapon.?,"+"]
tagabbrevs[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names. Example: result = this.tagabbrevs[Weapon.?,"+"]
tagids[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the unique ids for all matching tags instead of their names. Example: result = this.tagids[Weapon.?,"+"]
prereqok	(Right, Number) Returns non-zero if the pick satisfies all of its pre-requisites. May not be used during evaluation. Example: result = this.prereqok
prereqnum	(Right, Number) Returns the total number of pre-requisite rules that exist for the current pick. May not be used during evaluation. Example: result = this.prereqnum
prereqmsg[*index*]	(Right, String) Returns the message associated with a specific pre-requisite test for the current pick, where the desired test is given by *index*. The index is zero-based, so the values 0 through 4 should be used for a pick with 5 pre-requisites. If the individual pre-requisite is satisfied, the text returned is the empty string. Otherwise, the text returned is the corresponding message. May not be used during evaluation. Example: result = this.prereqmsg[i] **NOTE!** Retrieving the message for individual pre-requisite tests will trigger a re-calculation **every** time, so it is expensive and should only be used sparingly.
isgear	(Right, Number) Returns non-zero if the pick is a piece of gear. Example: result = this.isgear
isholdable	(Right, Number) Returns non-zero if the pick is gear that can be placed into a gear holder. Example: result = this.isholdable
isgearheld	(Right, Number) Returns non-zero if the pick is a piece of gear that is currently being held within another piece of gear. Example: result = this.isgearheld
gearcount	(Right, Number) Returns the total number of pieces of gear that are currently being held within the pick. If the pick is not a gear holder, the count will always be zero. Example: result = this.gearcount
gearpath[*spl*]	(Right, String) Generates and returns the entire gear containment hierarchy for this piece of gear. The hierarchy starts with the topmost gear holder of this pick and works downward through the immediate holder of the pick, showing the sequence. The generated string lists all the holders by name, inserting the splicing string *spl* between each name if there is more than one. A piece of gear that is not held will return the empty string. Example: result = this.gearpath["+"]
isgearlist	(Right, Number) Returns non-zero if the pick is a gear holder and can contain a list of held gear, regardless of whether any gear is actually held. Example: result = this.isgearlist
gearlist[*spl*,*expr*]	(Right, String) Generates and returns a list of gear held within the pick, provided the pick is a gear holder. Only picks that are designated as gear and that are assigned to this pick as their holder are candidates for inclusion in the list. Each candidate piece of gear is compared against the tag expression *expr*, and only those that satisfy the tag expression are included in the final list. The generated string appends the names of all pieces of gear together, inserting the splicing string *spl* between each name if there is more than one. A pick that holds no gear matching the tag expression will return the empty string. Example: result = this.gearlist["+",!Equipment.Natural]
stackable	(Right, Number) Returns non-zero if the pick can be stacked with other equivalent picks. Example: result = this.stackable
	(Right, Number) Returns non-zero if the current pick directly bootstraps a pick with the

isbootstrap[*thing*]	specified id *thing*. Example: result = this.isbootstrap[thingid]
rootnames[*spl*]	(Right, String) Generates and returns a list of picks that bootstrap the current pick. The generated string appends the names of the root picks together, inserting the splicing string *spl* between each name if there is more than one. A pick with no root picks will return the empty string. Example: result = this.rootnames["+"]
islinkage[*link*]	(Right, Number) Returns non-zero if the current pick possesses a linkage with the specified id *link*. Example: result = this.islinkage[linkid]
autonomous	(Right, Number) Returns non-zero if the pick has **zero** other picks that are currently based upon it. Picks that have other picks based upon them are typically made non-deletable, so this target reference provides a means to detect that condition. Example: result = this.autonomous
creation	(Right, Number) Returns non-zero if the pick was added to the character during "creation" mode and zero if added during "advancement" mode. This makes it possible to validate options that are only seelctable during character creation. If advancement mode is not enabled for the game system, all picks are always added in creation mode. Example: result = this.creation
setfocus	(Right, Number) Establishes the current pick context as the new "pick focus". Thereafter, the script can use the "focus." initial context transition to directly access the pick that is setup as the focus. Always returns a value of zero. Example: perform this.setfocus
ispanel	(Right, Number) Returns non-zero is the pick has a panel linkage defined for it. This allows generic handling of panel linkages within component scripts when the actual panel linkage is optionally controlled by the thing. Example: result = this.ispanel
panelactive	(Right, Number) Returns non-zero if the current active tab panel is the one specified as a panel linkage for the current pick. If the pick has no designated panel linkage, zero is returned. Example: result = this.panelactive
sourcerefs[*spl*]	(Right, String) Generates and returns a list of all sources that the current pick is dependent upon. The generated string appends the names of the sources together, inserting the splicing string *spl* between each name if there is more than one. A pick with no source dependencies will return the empty string. Example: result = this.sourcerefs["+"]
uniqindex	(Right, Number) Returns an integer value that uniquely identifies the pick throughout the entire portfolio. This value is *not* guaranteed to be the same when the portfolio is reloaded. It is intended for use in differentiating picks within rules and should never be saved in any way or otherwise used as a persistent reference. Example: result = this.uniqindex
dynareg[*index*]	(Right, Number) Registers a dynamic linkage to the current pick context and assigns that linkage the unique index value *index*. Once registered, the current pick can be accessed from the "hero" context via the "dynalink" context transition. This makes it possible to dynamically setup access to a special pick throughout an actor. This target reference can only be utilized from within a Creation Script. Always returns a value of zero. Example: perform this.dynareg[42]
isminion	(Right, Number) Returns non-zero if the pick resides within a minion, else zero. Example: result = this.isminion
ismaster	(Right, Number) Returns non-zero if the pick resides within a master, else zero. Example: result = this.ismaster
	(Right, Number) Returns non-zero if the pick resides within an actor that possesses a minion with the specified *id*. The parameter can be omitted, in which case non-zero is returned only if the pick directly attaches a minion. If the pick is not within a master or the specified minion

hasminion[*id*]	does not exist, zero is returned. Example: result = this.hasminion[myminion] Example: result = this.hasminion

Category: Kit Reference

Thing Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "thing" context represents a thing that has not been added to the portfolio. The thing context is very similar to the pick context in behavior, with the only real difference being that the dynamic facets of picks don't exist for things, resulting in many valid actions for picks being inaccessible from things.

Context Transitions

A "thing" context is very similar to a "pick" context, except that a "thing" context represents a thing that has not yet been added to a container and therefore lacks any dynamic state. As a result, the "thing" context is much more restrictive than the "pick" context. From within a "thing" context, you can utilize the following set of valid context transitions:

state	Transitions to the state context. Example: this.state
field[*id*]	Transitions to the field context corresponding to the field within the current thing that has the *id* specified. If the given field does not exist for the current thing, the transition fails to resolve. Example: this.field[myfield] **NOTE!** Within things and picks, there are a number of pre-defined pseudo-fields that are always defined and that allow access to facets of the pick that are not governed by user-defined fields. These pseudo-fields behave like normal fields in all respects within scripts, except that some are read-only. The list of pre-defined fields can be found elsewhere in the Kit Reference documentation.
linkage[*id*]	Transitions to the pick context corresponding to the linkage with the *id* specified. If the linkage is not defined, an error is reported. Example: this.linkage[mylink]

Target References

There are a number of target references that apply to things that have not been added to a container. These come into play at times like testing pre-requisites and when things are selected via menus. While similar to picks in many ways, things have no dynamic facets and are therefore always read-only in behavior. The complete list of target references for thing is presented in the table below.

idstring	(Right, String) Returns the unique id of the thing as a string. This can be extremely handy when synthesizing tag templates and tag expressions on-the-fly via scripts. Example: result = this.idstring
ispick	(Right, Number) Returns non-zero if the current context is a pick and zero if it's a thing. This provides a means to discern the nature of the context when the circumstances make a distinction uncertain, such as within procedures. Example: result = this.ispick
isunique	(Right, Number) Returns non-zero if the thing behaves as unique. Example: result = this.isunique
isentity	(Right, Number) Returns non-zero if the thing directly attaches a child entity. Example: result = this.isentity
tagis[*tmpl*]	(Right, Number) Returns non-zero if any tags assigned to the thing match the tag template *tmpl*, else zero if no tags match. The template must use the standard "group.id" syntax and may contain a wildcard. Example: this.tagis[skill.?]
tagcount[*tmpl*]	(Right, Number) Returns the number of tags assigned to the thing that match the tag template *tmpl*. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagcount[skill.?]
tagvalue[*tmpl*]	(Right, Number) Returns the value of a tag assigned to the thing that matches the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagvalue[spelllevel.wizard?]

tagmin[*tmpl*]	(Right, Number) Returns the minimum value of all tags assigned to the thing that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmin[spelllevel.wizard?]
tagmax[*tmpl*]	(Right, Number) Returns the maximum value of all tags assigned to the thing that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmax[spelllevel.wizard?]
tagunique[*tmpl*]	(Right, Number) Returns the number of unique tags assigned to the thing that match the tag template *tmpl*. If a tag is assigned multiple times, it is only counted once. The template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagunique[skill.?]
tagexpr[*expr*]	(Right, Number) Returns non-zero if the tags assigned to the thing match the tag expression *expr*, else zero is returned. The *expr* parameter must be a valid tag expression. Example: result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]
tagcountstr[*str*]	(Right, Number) This target reference is identical to "tagcount", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagcountstr["skill.?"]
tagvaluestr[*str*]	(Right, Number) This target reference is identical to "tagvalue", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagvaluestr["spelllevel.wizard?"]
tagminstr[*str*]	(Right, Number) This target reference is identical to "tagmin", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagminstr["spelllevel.wizard?"]
tagmaxstr[*str*]	(Right, Number) This target reference is identical to "tagmax", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagmaxstr["spelllevel.wizard?"]
taguniquestr[*str*]	(Right, Number) This target reference is identical to "tagunique", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.taguniquestr["skill.?"]
tagsearch[*str*]	(Right, Number) This target reference is identical to "tagexpr", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag expression to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagsearch["class.wizard & (val:spelllevel.wizard? > 5)"] **NOTE!** Using "tagsearch" is MUCH slower in performance than using "tagexpr", since the tagexpr must be parsed and compiled every time the script is evaluated. Be sure to use "tagexpr" whenever possible.
tagmatch[*refgrp*,*match*,*ref*]	(Right, Number) Returns non-zero if any tags within a reference context also exist within a match context, else zero. This allows tags from one context to be verified to exist within another context, making tag-based comparisons between two objects possible. For more details, please check here. Example: result = hero.tagmatch[NeedStatus,HasStatus,initial]
intersect[*init*,*curr*]	(Right, Number) Returns non-zero if tags within the initial script context also exist within the currently transitioned context, else zero. All of the tags that belong to the *init* tag group within the initial script context are compared against the tags that belong to the *curr* tag group within the current context. If any of those tags possess the identical tag id, then a match is returned (i.e. non-zero). The same tag group may be used for both contexts, and both contexts must be objects that possess tags (i.e. container, pick, or thing). Example: result = this.interset[MyGroup,AltGroup]

tagnames[*tmpl*,*spl*]	(Right, String) Generates and returns a list of tags within the thing. Only tags that match the tag template *tmpl* are included in the list. The generated string appends the names of the tags together, inserting the splicing string *spl* between each name if there is more than one. A thing with no tags matching the template will return the empty string. Example: result = this.tagnames[Weapon.?,"+"]
tagabbrevs[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the abbreviations for all matching tags instead of their names. Example: result = this.tagabbrevs[Weapon.?,"+"]
tagids[*tmpl*,*spl*]	(Right, String) Works identically to "tagnames", except that the generated string is comprised of the unique ids for all matching tags instead of their names. Example: result = this.tagids[Weapon.?,"+"]
prereqok	(Right, Number) Returns non-zero if the pick satisfies all of its pre-requisites. May not be used during evaluation. Example: result = this.prereqok
prereqnum	(Right, Number) Returns the total number of pre-requisite rules that exist for the current pick. May not be used during evaluation. Example: result = this.prereqnum
prereqmsg[*index*]	(Right, String) Returns the message associated with a specific pre-requisite test for the current pick, where the desired test is given by *index*. The index is zero-based, so the values 0 through 4 should be used for a pick with 5 pre-requisites. If the individual pre-requisite is satisfied, the text returned is the empty string. Otherwise, the text returned is the corresponding message. May not be used during evaluation. Example: result = this.prereqmsg[i] **NOTE!** Retrieving the message for individual pre-requisite tests will trigger a re-calculation **every** time, so it is expensive and should only be used sparingly.
isgear	(Right, Number) Returns non-zero if the thing is a piece of gear. Example: result = this.isgear
isholdable	(Right, Number) Returns non-zero if the pick is gear that can be placed into a gear holder. Example: result = this.isholdable
stackable	(Right, Number) Returns non-zero if the thing can be stacked with other equivalent picks. Example: result = this.stackable
isbootstrap[*thing*]	(Right, Number) Returns non-zero if the current thing directly bootstraps a thing with the specified id *thing*. Example: result = this.isbootstrap[thingid]
islinkage[*link*]	(Right, Number) Returns non-zero if the current thing possesses a linkage with the specified id *link*. Example: result = this.islinkage[linkid]
ispanel	(Right, Number) Returns non-zero if the thing has a panel linkage defined for it. This allows generic handling of panel linkages within component scripts when the actual panel linkage is optionally controlled by the thing. Example: result = this.ispanel
sourcerefs[*spl*]	(Right, String) Generates and returns a list of all sources that the current thing is dependent upon. The generated string appends the names of the sources together, inserting the splicing string *spl* between each name if there is more than one. A thing with no source dependencies will return the empty string. Example: result = this.sourcerefs["+"]
	(Right, String) Modifies the name or description of the thing henceforth for the current actor. Once amended, all subsequent references to the thing will return the new contents. This includes the details shown within chooser tables that display things for selection by the user. The *targ* parameter must be either "name" or "description", specifying which facet of the thing is being amended. The new contents are given by the string expression *str*. Always returns a value of zero. Example: result = this.amendthing[name,"New Name"]

amendthing[*targ*,*str*]	**NOTE!** The amendment is applied to the thing at the time of evaluation, so any access of the name or description on derived picks prior to the amendment will not show the effects of the amendment. All amendments persist until the start of the next evaluation cycle, at which time they are all reset. **NOTE!** Each actor maintains its own set of amendments, so it is perfectly valid to have multiple different actors with different amendments to the same thing.

Category: Kit Reference

Field Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "field" context represents any field within any pick or thing. If within a thing, all aspects of the field are read-only.

Context Transitions

From within a "field" context, you can utilize the following set of valid context transitions:

state	Transitions to the state context. Example: this.state
pick	Transitions to the pick context corresponding to the pick that contains the current field. Example: this.pick
chosen	Transitions to either the pick context corresponding to the user-selected pick stored within the field or the thing context corresponding to the user-selected thing stored within the field, depending on the nature of the field. The transition is only valid for use on fields that store menu selections made by the user. Example: this.chosen

Target References

There are a variety of ways to access and manipulate the contents of fields. The complete list of target references for fields is presented in the table below.

NOTE! When a field is accessed from within a thing or a visual script, all operations are restricted to read-only behavior. Any attempts to modify a field from within a thing context or via a visual script will fail.

value	(Left, Right, Number) Accesses the contents of the field as a numeric value. If the field is text and the contents retrieved, the contents are automatically converted to a suitable value. Example: result = this.field[myfield].value Example: this.field[myfield].value = 42
text	(Left, Right, String) Accesses the contents of the field as a string. If the field is numeric can the contents retrieved, the contents are automatically converted to a suitable string. Example: result = this.field[myfield].text Example: this.field[myfield].text = "hello"
isempty	(Right, Number) Returns non-zero if the field contains the empty string and zero if it contains text of any length. If used on a numeric field or with an array or matrix, an error is reported. Testing of array and matrix elements must be done via the "compare" intrinsic. Example: result = this.field[myfield].isempty
ischanged	(Right, Number) Returns non-zero if the field contents have been changed in any way from its original starting state. This can detect a field that has been changed by the user or via a script. Example: result = this.field[myfield].ischanged
reset	(Right, Number) The contents of the field are reset to the initial default value for the thing. A value of zero is always returned. Example: perform this.field[myfield].reset
ischosen	(Right, Number) Returns non-zero if the field contains a pick or a thing that was selected via a menu. If the field is associated with a menu and no selection has yet been made, zero is returned. Example: result = this.field[myfield].ischosen
delta	(Left, Right, Number) Accesses the delta component of a user field for manipulation separately from the user-specified value. The field must be explicitly configured to support delta processing. Example: result = this.field[myfield].delta Example: this.field[myfield].delta = 2
	(Left, Right, Number) Accesses the contents of a specific element of the array-based field as a numeric value. The index of the element is given by *row*, where the index is a zero-based value that must be less than the number of rows in the array. If the field is text and the contents retrieved, the contents

arrayvalue[*row*]

are automatically converted to a suitable value.
Example: result = this.field[myfield].arrayvalue[3]
Example: this.field[myfield].arrayvalue[3] = 42

arraytext[*row*]

(Left, Right, Number) Accesses the contents of a specific element of the array-based field as a string. The index of the element is given by *row*, where the index is a zero-based value that must be less than the number of rows in the array. If the field is numeric can the contents retrieved, the contents are automatically converted to a suitable string.
Example: result = this.field[myfield].arraytext[3]
Example: this.field[myfield].arraytext[3] = "hello"

matrixvalue[*row*,*col*]

(Left, Right, Number) Accesses the contents of a specific element of the matrix-based field as a numeric value. The index of the element is given by *row* and *col*, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is text and the contents retrieved, the contents are automatically converted to a suitable value.
Example: result = this.field[myfield].matrixvalue[3,4]
Example: this.field[myfield].matrixvalue[3,4] = 42

matrixtext[*row*,*col*]

(Left, Right, String) Accesses the contents of a specific element of the matrix-based field as a string. The index of the element is given by *row* and *col*, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is numeric and the contents retrieved, the contents are automatically converted to a suitable string.
Example: result = this.field[myfield].matrixvalue[3,4]
Example: this.field[myfield].matrixvalue[3,4] = 42

arraydump

(Right, String) Generates and returns a text string that contains the values of all elements in the array, with the results appropriately formatted for easy viewing. This mechanism is ideal for use within "debug" statements to help isolate scripting problems with arrays. Only usable with array-based fields.
Example: result = this.field[myfield].arraydump
NOTE! The generated output is capped to the limits of the debug output mechanism, so using this mechanism with large arrays and/or text-based fields may result in the output being truncated.

matrixdump

(Right, String) Generates and returns a text string that contains the values of all elements in the matrix, with the results appropriately formatted for easy viewing. This mechanism is ideal for use within "debug" statements to help isolate scripting problems with matrices. Only usable with matrix-based fields.
Example: result = this.field[myfield].matrixdump
NOTE! The generated output is capped to the limits of the debug output mechanism, so using this mechanism with large matrices and/or text-based fields may result in the output being truncated.

datetime[*fmt*,*sep*]

(Right, String) Returns the contents of the field formatted for output as a date or time, where *sep* is the separator string to use between values. The *fmt* parameter dictates the formatting to be used and must be one of the following values:

- realdate – Formatted as if it's a real-world date.
- realtime – Formatted as if it's a real-world time.
- gamedate – Formatted as if it's a game-world date.
- gametime – Formatted as if it's a game-world time.

Example: result = this.field[myfield].datetime[gamedate,"/"]
NOTE! This target reference is only supported on fields within picks - not things.

(Right, Number) Applies a modification to the field with accompanying notes for tracking a history of changes. The modification nature is given by the *oper* parameter, which which specifies the operation to be performed. The *val* parameter is an arithmetic expression the provides the modification value to be applied. The *str* parameter is a text annotation that is recorded along with the modification. If *str* is the empty string, the notes are the name of the initial pick context that is applying the change (or "???
??" if the initial context is not a pick). The *oper* parameter must be one of the following:

- '+' - The value is added to the field.
- '-' - The value is subtracted from the field.
- '*' - The value is multiplied into the field.

modify[*oper*,*val*,*str*]

'/' - The value is divided into the field.

- '=' - The new value is assigned to the field and all previous adjustments to the field are ignored. This operator is only valid with "stack" or "changes" history tracking.
- '#' - The value is added to the field, but no sign is displayed for the field within the history report. This allows the identification of basic values included into the field calculation that are distinct from bonuses (which include the '+').
- '$' - The value is completely ignored and only the text entry is added for the modification. This allows a history entry to be recorded that has no value but that needs to be included in the report.

The field must be explicitly configured to support history tracking. Always returns a value of zero.
Example: perform this.field[myfield].modify[+,1,"first"]

history[*spl*,*start*]

(Right, String) Generates and returns a text string that contains the change history for the field. The *spl* parameter is a string that is inserted between entries in the change history, splicing them together for appropriate output. The *start* parameter is the starting value used for the field within the report. Both the *spl* and *start* parameters are optional, although the *spl* is required in order to specify the *start* parameter. If omitted, the *spl* parameter defaults to a comma and a space (", "), while the *start* parameter defaults to zero. The history report contents depend on the history behavior assigned to the field, as given below:

- best – Only the notes text for each history entry is reported
- stack – The notes text for each entry is reported and the adjustment details are included in parentheses with the notes (e.g. "notes (+2)")
- changes - Same as 'stack'

Example: result = this.field[myfield].history
Example: result = this.field[myfield].history["&"]
Example: result = this.field[myfield].history[", ",42]

Category: Kit Reference

Pool Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "pool" context represents any usage pool associated with either the actor or a specific pick.

Context Transitions

From within a "pool" context, you can utilize the following set of valid context transitions:

-None-	There are no transitions from within a pool context.

Target References

The "pool" script context corresponds to the contents of a usage pool, whether it be associated with a pick or directly on the actor. The complete list of target references for usage pools is presented in the table below.

name	(Right, String) Returns the name assigned to the usage pool. Example: result = this.name
abbrev	(Right, String) Returns the abbreviation assigned to the usage pool. Example: result = this.abbrev
count	(Right, Number) Returns the number of historical entries that currently exist within the usage pool. Example: result = this.count
value	(Right, Number) Returns the net adjusted value for the usage pool, after applying all of the adjustments that have been assigned. Example: result = this.value
empty	(Right, Number) Discards all adjustments for the usage pool and resets the value to its initial default. Always returns a value of zero. Example: perform this.empty
adjust[*value*]	(Right, Number) Applies an adjustment to the usage pool in the amount given by the *value* parameter, which can be an arithmetic expression of any type. Always returns a value of zero. Example: perform this.adjust[42]
set[*value*]	(Right, Number) Applies an adjustment to the usage pool that sets the net value of the pool to the new total given by the *value* parameter. For example, if the current net value of the pool is 14 and *value* is 17, an adjustment of 3 is applied. Always returns a value of zero. Example: perform this.set[42]
rollback	(Right, Number) Rolls back (i.e. undoes) the most recent adjustment that was applied to the usage pool. Always returns a value of zero. Example: perform this.rollback
history[*index*]	(Right, Number) Returns the value of an individual adjustment in the change history. The specific entry is given by the *index* parameter, which can be an arithmetic expression. The index is zero-based, so the index must a value between zero and the total number of entries in the usage pool minus one. The zeroth entry is the most recent adjustment, with an increasing value proceeding further backwards in history. This provides a way to retrieve a report of all of the adjustments in the history of the pool and format them however you want. Example: result = this.history[3]

Category: Kit Reference

Scene Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "scene" context identifies the top-level visual element within the current hierarchy, which will be either a panel, a form, or a sheet.

Context Transitions

From within a "scene" context, you can utilize the following set of valid context transitions:

layout[*id*]	Transitions to the layout context corresponding to the layout within the current scene that possesses the *id* specified. If the layout does not exist within the scene, the transition fails to resolve. Example: this.layout[mylayout]
container	Transitions to the container context corresponding to the container to which the scene applies, whether it be an actor or a gizmo. Example: this.container **NOTE!** After transitioning, access within the new container context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.
hero	Transitions to the hero context corresponding to the hero to which the scene applies. Example: this.hero **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.

Target References

The "scene" script context applies equally to panels, forms, and sheets. However, there are some important behavioral differences between those three visual elements that impact how certain target references operate for each, and those differences are detailed below. The complete list of target references for scenes is presented in the table below.

width	(Left, Right, Number) Accesses the width of the scene. The width of panels and sheets is properly setup by HL and any changes are completely ignored, as HL wholly controls the rendering region for panels and sheets. For forms, the width is initialized to something safe by HL, but the author is assumed to set the width appropriately for the contents that need to be displayed. Example: result = this.width Example: this.width = 420
height	(Left, Right, Number) Accesses the height of the scene. The same rules apply as for the "width" target reference above. Example: result = this.height Example: this.height = 420
scrollbar	(Right, Number) Returns the width of a scroller, in pixels. Example: result = this.scrollbar
defwidth	(Right, Number) Returns the default width for a form that is specified within the form's definition. Only applicable to forms. Example: result = this.defwidth
defheight	(Right, Number) Returns the default height for a form that is specified within the form's definition. Only applicable to forms. Example: result = this.defheight
minwidth	(Left, Right, Number) Accesses the minimum width governing the form. A value of zero indicates the minimum width should be the default width. If both minwidth and maxwidth are zero, the form cannot be resized by the user. If data files specify both minwidth and maxwidth as zero, the form width cannot be modified via scripts. If the min/max values are modified such that the current dimensions of the form become invalid, the form dimensions are automatically adjusted to comply with the new limits. Only applicable to forms. Example: result = this.minwidth Example: this.minwidth = 420

minheight	(Left, Right, Number) Accesses the minimum height governing the form. The same rules apply as for the "minwidth" target reference above. Example: result = this.minheight Example: this.minheight = 420
maxwidth	(Left, Right, Number) Accesses the maximum width governing the form. The same rules apply as for the "minwidth" target reference above. Example: result = this.maxwidth Example: this.maxwidth = 420
maxheight	(Left, Right, Number) Accesses the maximum height governing the form. The same rules apply as for the "minwidth" target reference above. Example: result = this.maxheight Example: this.maxheight = 420

Category: Kit Reference

Layout Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "layout" context identifies a layout within the top-level scene of the current hierarchy.

Context Transitions

From within a "layout" context, you can utilize the following set of valid context transitions:

parent	Transitions to the scene context corresponding to the visual element that contains the layout. Example: this.parent
template[*id*]	Transitions to the template context corresponding to the template within the current layout that possesses the *id* specified. If the template does not exist within the layout, the transition fails to resolve. Example: this.template[mytemplate] **NOTE!** The id specified is the id of the template **reference**, as defined within the layout, and not necessarily the id of the template itself.
portal[*id*]	Transitions to the portal context corresponding to the portal within the current layout that possesses the *id* specified. If the portal does not exist within the layout, the transition fails to resolve. Example: this.portal[myportal] **NOTE!** The id specified is the id of the portal **reference**, as defined within the layout, and not necessarily the id of the portal itself. **NOTE!** Only portals defined directly within the layout can be accessed via this transition. Portals within templates must be accessed via the template.
container	Transitions to the container context corresponding to the container to which the layout applies, whether it be an actor or a gizmo. Example: this.container **NOTE!** After transitioning, access within the new container context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.
hero	Transitions to the hero context corresponding to the hero to which the layout applies. Example: this.hero **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.

Target References

The "layout" script context governs the operations that can be applied to layouts within scenes. The complete list of target references for layouts is presented in the table below.

width	(Left, Right, Number) Accesses the width of the layout. Unless explicitly specified within the XML, the width of a layout is initialized to the width of the containing scene, minus any assigned margins. Example: result = this.width Example: this.width = 420
height	(Left, Right, Number) Accesses the height of the scene. Unless explicitly specified within the XML, the height of a layout is initialized to the height of the containing scene, minus any assigned margins. Example: result = this.height Example: this.height = 420
left	(Left, Right, Number) Accesses the position of the left edge of the layout within the containing visual element. Example: result = this.left Example: this.left = 42
	(Left, Right, Number) Accesses the position of the top edge of the layout within the containing visual element.

top	Example: result = this.top Example: this.top = 42
right	(Right, Number) Returns the position of the right edge of the layout within the containing visual element. Example: result = this.right
bottom	(Right, Number) Returns the position of the bottom edge of the layout within the containing visual element. Example: result = this.bottom
visible	(Left, Right, Number) Controls the visibility of the layout within the containing visual element. A non-zero value indicates the layout is visible and a zero value indicates hidden. Example: result = this.visible Example: this.visible = 1
scrollbar	(Right, Number) Returns the width of a scroller, in pixels. Example: result = this.scrollbar
autotop	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element. Example: result = this.autotop Example: this.autotop = 42
autobottom	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element. Example: result = this.autobottom Example: this.autobottom = 420
autoleft	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element. Example: result = this.autoleft Example: this.autoleft = 42
autoright	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element. Example: result = this.autoright Example: this.autoright = 420
autowidth	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element. Example: result = this.autowidth Example: this.autowidth = 420
autoheight	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element. Example: result = this.autoheight Example: this.autoheight = 420
autogap	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero. Example: result = this.autogap Example: this.autogap = 42
autoplace[*gap*]	(Right, Number) Automatically places the layout within the containing visual element, subject to the standard rules for automatic placement. The *gap* parameter specifies the gap to be used between this layout and the previous placed element. The parameter can be omitted, in which case the established "autogap" is utilized. A value of zero is always returned. Example: perform this.autoplace[42] Example: perform this.autoplace

Category: Kit Reference

Template Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "template" context identifies a template within the current hierarchy. Templates can either be used within layouts or within tables, so the parent context of the template within the hierarchy can vary.

Context Transitions

From within a "template" context, you can utilize the following set of valid context transitions:

parent	Transitions to the layout context or table context corresponding to the visual element that contains the tmeplate. Example: this.parent **NOTE!** The "parent" transition can only be utilized a **single** time, so it is not possible to go upwards two or more levels within the hierarchy.
portal[*id*]	Transitions to the portal context corresponding to the portal within the current template that possesses the *id* specified. If the portal does not exist within the template, the transition fails to resolve. Example: this.portal[myportal] **NOTE!** Only portals defined directly within the template can be accessed via this transition.
field[*id*]	Transitions to the value context corresponding to the field within the current template that has the *id* specified. The fields for a template are dictated by the pick or thing that is associated with the template. If the given field does not exist for the pick/thing, the transition fails to resolve. Example: this.field[myfield] **NOTE!** Within templates, all fields are treated as read-only, which is controlled by transitioning to a distinct "value" context instead of "field" context.
container	Transitions to the container context corresponding to the container to which the template applies, whether it be an actor or a gizmo. Example: this.container **NOTE!** After transitioning, access within the new container context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.
hero	Transitions to the hero context corresponding to the hero to which the template applies. Example: this.hero **NOTE!** After transitioning, access within the new hero context will be read-only and limited in what information can be retrieved. **NOTE!** This transition can only be used as the **first** transition when within a visual script.

Target References

The "template" script context governs the operations that can be applied to templates within layouts and tables. The complete list of target references for templates is presented in the table below.

width	(Left, Right, Number) Accesses the width of the template. Unless explicitly specified within the XML, the width of a template is automatically initialized. If the template is within a table, the width is set the interior width of the containing table, minus any assigned margins. Otherwise, the template is initialized to a width of 100. Example: result = this.width Example: this.width = 420
height	(Left, Right, Number) Accesses the height of the template. Unless explicitly specified within the XML, the height of a layout is automatically initialized to zero and must be set by the author via a suitable script. Example: result = this.height Example: this.height = 420
left	(Left, Right, Number) Accesses the position of the left edge of the template within the containing visual element. Example: result = this.left Example: this.left = 42

top	(Left, Right, Number) Accesses the position of the top edge of the template within the containing visual element. Example: result = this.top Example: this.top = 42
right	(Right, Number) Returns the position of the right edge of the template within the containing visual element. Example: result = this.right
bottom	(Right, Number) Returns the position of the bottom edge of the template within the containing visual element. Example: result = this.bottom
visible	(Left, Right, Number) Controls the visibility of the template within the containing visual element. A non-zero value indicates the template is visible and a zero value indicates hidden. The visibility of templates cannot be controlled within tables. Example: result = this.visible Example: this.visible = 1
isroot	(Right, Number) Returns non-zero if the pick associated with the template has been bootstrapped and therefore has a root pick available. If the pick is associated with a thing, zero is returned. Example: result = this.isroot
issizing	(Right, Number) Returns non-zero if the "sizing" logic of a template within a table is being performed. This allows detection of the sizing logic so that all non-sizing behaviors can be skipped for the template. Example: result = this.issizing
inheader	(Right, Number) Returns non-zero if the template is being positioned for use as a header within a table. This makes it possible to distinguish the context for a template that is being used as a dual-purpose header. Example: result = this.inheader
scrollbar	(Right, Number) Returns the width of a scroller, in pixels. Example: result = this.scrollbar
tagis[*tmpl*]	(Right, Number) Returns non-zero if any tags assigned to the pick/thing associated with the template that match the tag template *tmpl*, else zero if no tags match. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: this.tagis[skill.?]
tagcount[*tmpl*]	(Right, Number) Returns the number of tags assigned to the pick/thing associated with the template that match the tag template *tmpl*. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagcount[skill.?]
tagvalue[*tmpl*]	(Right, Number) Returns the value of a tag assigned to the pick/thing associated with the template that matches the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagvalue[spelllevel.wizard?]
tagmin[*tmpl*]	(Right, Number) Returns the minimum value of all tags assigned to the pick/thing associated with the template that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmin[spelllevel.wizard?]
tagmax[*tmpl*]	(Right, Number) Returns the maximum value of all tags assigned to the pick/thing associated with the template that match the tag template *tmpl*. The rules associated with tag values in tag expressions apply. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagmax[spelllevel.wizard?]
tagunique[*tmpl*]	(Right, Number) Returns the number of unique tags assigned to the pick/thing associated with the template that match the tag template *tmpl*. If a tag is assigned multiple times, it is only counted once. The tag template must use the standard "group.id" syntax and may contain a wildcard. Example: result = this.tagunique[skill.?]
tagexpr[*expr*]	(Right, Number) Returns non-zero if the tags assigned to the pick/thing associated with the template match the tag expression *expr*, else zero is returned. The *expr* parameter must be a valid tag expression. Example: result = this.tagexpr[class.wizard & (val:spelllevel.wizard? > 5)]

tagcountstr[*str*]	(Right, Number) This target reference is identical to "tagcount", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagcountstr["skill.?"]
tagvaluestr[*str*]	(Right, Number) This target reference is identical to "tagvalue", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagvaluestr["spelllevel.wizard?"]
tagminstr[*str*]	(Right, Number) This target reference is identical to "tagmin", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagminstr["spelllevel.wizard?"]
tagmaxstr[*str*]	(Right, Number) This target reference is identical to "tagmax", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.tagmaxstr["spelllevel.wizard?"]
taguniquestr[*str*]	(Right, Number) This target reference is identical to "tagunique", except that the *str* parameter is a string expression that is evaluated within the script. This allows the tag template to be dynamically determined via the script instead of being hard-wired at compilation. Example: result = this.taguniquestr["skill.?"]
isidentity[*grp*]	(Right, Number) Returns non-zero if the indicated identity tag has been assigned to the pick/thing associated with the template. The identity tag sought must be from the tag group *grp* and the tag id is dictated by the initial context of the script. For more details, please check here. Example: result = this.isidentity[groupid]
autotop	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element. Example: result = this.autotop Example: this.autotop = 42
autobottom	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element. Example: result = this.autobottom Example: this.autobottom = 420
autoleft	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element. Example: result = this.autoleft Example: this.autoleft = 42
autoright	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element. Example: result = this.autoright Example: this.autoright = 420
autowidth	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element. Example: result = this.autowidth Example: this.autowidth = 420
autoheight	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element. Example: result = this.autoheight Example: this.autoheight = 420
autogap	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero. Example: result = this.autogap Example: this.autogap = 42
	(Right, Number) Automatically places the template within the containing layout, subject to the standard rules for automatic placement. The *gap* parameter specifies the gap to be used between this template and

autoplace[*gap*]	the previous placed element. The parameter can be omitted, in which case the established "autogap" is utilized. Automatic placement for templates can only be used within layouts. A value of zero is always returned. Example: perform this.autoplace[42] Example: perform this.autoplace

Category: Kit Reference

Portal Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "portal" context identifies a portal within the current hierarchy. Portals can either be used within templates or within layouts, so the parent context of the portal within the hierarchy can vary.

Context Transitions

From within a "portal" context, you can utilize the following set of valid context transitions:

parent	Transitions to the layout context or template context corresponding to the visual element that contains the portal. Example: this.parent **NOTE!** The "parent" transition can only be utilized a **single** time, so it is not possible to go upwards two or more levels within the hierarchy.

Target References

The "portal" script context governs the operations that can be applied to portals within templates and layouts. The complete list of target references for portals is presented in the table below.

width	(Left, Right, Number) Accesses the width of the portal. Unless explicitly specified within the XML, the width of a portal is automatically initialized appropriately to the nature of the portal. Setting the width of a table portal triggers immediate re-sizing of all items within the table. Example: result = this.width Example: this.width = 42
height	(Left, Right, Number) Accesses the height of the portal. Unless explicitly specified within the XML, the height of a portal is automatically initialized appropriately to the nature of the portal. Setting the height of a table portal triggers immediate re-sizing of all items within the table. Example: result = this.height Example: this.height = 42
left	(Left, Right, Number) Accesses the position of the left edge of the portal within the containing visual element. Example: result = this.left Example: this.left = 42
top	(Left, Right, Number) Accesses the position of the top edge of the portal within the containing visual element. Example: result = this.top Example: this.top = 42
right	(Right, Number) Returns the position of the right edge of the portal within the containing visual element. Example: result = this.right
bottom	(Right, Number) Returns the position of the bottom edge of the portal within the containing visual element. Example: result = this.bottom
visible	(Left, Right, Number) Controls the visibility of the portal within the containing visual element. A non-zero value indicates the portal is visible and a zero value indicates hidden. Example: result = this.visible Example: this.visible = 1
freeze	(Left, Right, Number) Controls whether the portal is displayed in a "frozen" state, where a frozen portal is non-editable and for display only. A non-zero value indicates the portal is frozen and a zero value indicates normal behavior. Example: result = this.freeze Example: this.freeze = 1 **NOTE!** All portals must be frozen **before** any portals are positioned.
scrollbar	(Right, Number) Returns the width of a scroller, in pixels. Example: result = this.scrollbar

sizetofit[*min*]	(Right, Number) Reduces the font size, as necessary, to fit the text contents of the portal into the available space given by the portal dimensions. The font size will be reduced in quarter-point increments until either the text fits or the minimum font size given by *min* is reached. The value zero is always returned. Example: result = this.sizetofit[36] **NOTE!** This target reference is only valid for use with the various types of label portals. **NOTE!** The minimum font size allowed is 6-point for screen output and 4-point for printed output.
autotop	(Left, Right, Number) Accesses the position of the top edge of the auto-place region within the containing visual element. Example: result = this.autotop Example: this.autotop = 42
autobottom	(Left, Right, Number) Accesses the position of the bottom edge of the auto-place region within the containing visual element. Example: result = this.autobottom Example: this.autobottom = 420
autoleft	(Left, Right, Number) Accesses the position of the left edge of the auto-place region within the containing visual element. Example: result = this.autoleft Example: this.autoleft = 42
autoright	(Left, Right, Number) Accesses the position of the right edge of the auto-place region within the containing visual element. Example: result = this.autoright Example: this.autoright = 420
autowidth	(Left, Right, Number) Accesses the width of the auto-place region within the containing visual element. Example: result = this.autowidth Example: this.autowidth = 420
autoheight	(Left, Right, Number) Accesses the height of the auto-place region within the containing visual element. Example: result = this.autoheight Example: this.autoheight = 420
autogap	(Left, Right, Number) Accesses the default gap size used when automatically placing elements within the containing visual element. The "autogap" defaults to zero. Example: result = this.autogap Example: this.autogap = 42
autoplace[*gap*]	(Right, Number) Automatically places the portal within the containing layout, subject to the standard rules for automatic placement. The *gap* parameter specifies the gap to be used between this portal and the previously placed element. The parameter can be omitted, in which case the established "autogap" is utilized. Automatic placement for portals can only be used within layouts. A value of zero is always returned. Example: perform this.autoplace[42] Example: perform this.autoplace

Category: Kit Reference

Table Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "table" context identifies the containing table of a template or portal within the current hierarchy.

Context Transitions

The "table" context is essentially the same as a "portal" context, since the context simply corresponds to a table portal within a layout. From within a "table" context, you can utilize the following set of valid context transitions:

parent	Transitions to the layout context corresponding to the visual element that contains the portal. Example: this.parent **NOTE!** The "parent" transition can only be utilized a **single** time, so it is not possible to go upwards two or more levels within the hierarchy.

Target References

Tables are a special type of portal. As such, they share all of the same target references as normal portals. However, tables also have a few additional target references that are unique to themselves. The complete list of these special target references is presented in the table below.

IMPORTANT! Tables also support all general portal target references.

gapx	(Right, Number) Returns the gap between items within the table along the X-axis. Example: perform this.gapx
gapy	(Right, Number) Returns the gap between items within the table along the Y-axis. Example: perform this.gapy

Category: Kit Reference

Value Context

Context: HL Kit … Kit Reference … Multiple Sources

Jump to: Target References

The "value" context represents any field within the pick or thing associated with a template. The value context is used for display only, so all aspects of the field are always read-only.

Context Transitions

The "value" context parallels a "field" context, with the key distinction being that all values are read-only, since they are intended for display only. From within a "value" context, you can utilize the following set of valid context transitions:

-None-	There are no transitions from within a value context.

Target References

The "value" script context is actually the same as a "field" context, except that the field is being accessed from within a visual script and is therefore read-only in all behaviors. Hence the need to keep the script contexts distinct. The complete list of target references for the "value" context is presented in the table below.

value	(Right, Number) Returns the contents of the field as a numeric value. If the field is text, the contents are automatically converted to a suitable value. Example: result = this.field[myfield].value
text	(Right, String) Returns the contents of the field as a string. If the field is numeric, the contents are automatically converted to a suitable string. Example: result = this.field[myfield].text
isempty	(Right, Number) Returns non-zero if the field contains the empty string and zero if it contains text of any length. If used on a numeric field or with an array or matrix, an error is reported. Testing of array and matrix elements must be done via the "compare" intrinsic. Example: result = this.field[myfield].isempty
ischanged	(Right, Number) Returns non-zero if the field contents have been changed in any way from its original starting state. This can detect a field that has been changed by the user or via a script. Example: result = this.field[myfield].ischanged
ischosen	(Right, Number) Returns non-zero if the field contains a pick or a thing that was selected via a menu. If the field is associated with a menu and no selection has yet been made, zero is returned. Example: result = this.field[myfield].ischosen
delta	(Right, Number) Returns the delta component of a user field. The field must be explicitly configured to support delta processing. Example: result = this.field[myfield].delta
arrayvalue[*row*]	(Right, Number) Returns the contents of a specific element of the array-based field as a numeric value. The index of the element is given by *row*, where the index is a zero-based value that must be less than the number of rows in the array. If the field is text, the contents are automatically converted to a suitable value. Example: result = this.field[myfield].arrayvalue[3]
arraytext[*row*]	(Right, Number) Returns the contents of a specific element of the array-based field as a string. The index of the element is given by *row*, where the index is a zero-based value that must be less than the number of rows in the array. If the field is numeric, the contents are automatically converted to a suitable string. Example: result = this.field[myfield].arraytext[3]
matrixvalue[*row*,*col*]	(Right, Number) Returns the contents of a specific element of the matrix-based field as a numeric value. The index of the element is given by *row* and *col*, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is text, the contents are automatically converted to a suitable value. Example: result = this.field[myfield].matrixvalue[3,4]
	(Right, String) Returns the contents of a specific element of the matrix-based field as a string. The

matrixtext[*row*,*col*]	index of the element is given by *row* and *col*, where the indices are zero-based values that must be less than the number of rows and columns in the matrix, respectively. If the field is numeric, the contents are automatically converted to a suitable string. Example: result = this.field[myfield].matrixvalue[3,4]
datetime[*fmt*,*sep*]	(Right, String) Returns the contents of the field formatted for output as a date or time, where *sep* is the separator string to use between values. The *fmt* parameter dictates the formatting to be used and must be one of the following values: ▪ realdate – Formatted as if it's a real-world date. ▪ realtime – Formatted as if it's a real-world time. ▪ gamedate – Formatted as if it's a game-world date. ▪ gametime – Formatted as if it's a game-world time. Example: result = this.field[myfield].datetime[gamedate,"/"] **NOTE!** This target reference is only supported on fields within picks - not things.
history[*spl*]	(Right, String) Generates and returns a text string that contains the change history for the field. The *spl* parameter is a string that is inserted between entries in the change history, splicing them together for appropriate output. The history report contents depend on the history behavior assigned to the field, as given below: ▪ best – Only the notes text for each history entry is reported ▪ stack – The notes text for each entry is reported and the adjustment details are included in parentheses with the notes (e.g. "notes (+2)") **NOTE!** This target reference is only supported on fields within picks - not things.

Category: Kit Reference

State Context

Context: HL Kit ... Kit Reference ... Multiple Sources

Jump to: Target References

The "state" context provides access to overall state information that pertains to the portfolio as a whole or general information about the evaluation cycle.

Context Transitions

From within a "state" context, you can utilize the following set of valid context transitions:

thing[*id*]	Transitions to the thing context corresponding to the thing with the *id* specified. If no thing exists with the given unique id, an error is reported. Example: this.thing[myid]

Target References

The "state" context is a general context that maintains information outside the normal data hierarchy. The target references for this context span a wide range of details that may prove useful within your scripts. The complete list of target references for the "state" context is presented in the table below.

isfocus	(Right, Number) Returns non-zero if a focus pick has been properly established via the "setfocus" target reference. Example: result = state.isfocus
clearfocus	(Right, Number) Clears any focus pick that has been established and resets to a state where there is no focus pick. Example: perform state.clearfocus
timing	(Right, String) Returns the phase and priority timing during which the current script is being invoked in an effort to simplify debugging of data files. Example: result = state.timing
iscreate	(Right, Number) Returns non-zero if the character is currently in creation mode. Example: result = state.iscreate
isadvance	(Right, Number) Returns non-zero if the character is currently in advancement mode. Example: result = state.isadvance
issell	(Right, Number) Returns non-zero if the user is currently in the midst of a sell transaction. Example: result = state.issell
isload	(Right, Number) Returns non-zero if the loading of a saved portfolio is currently in progress. Example: result = state.isload
isoutput	(Right, Number) Returns non-zero if the rendering of character sheet output is currently in progress. Example: result = state.isoutput
isdossierstyle[*style*]	(Right, Number) Returns whether the user selected text output to be formatted in the style given by the *style* parameter. The *style* parameter must be one of the following values: "plain", "html", or "bbcode". Example: result = state.isdossierstyle[html]
istext	(Right, Number) Returns non-zero if the render of text output is currently in progress. Example: result = state.istext
iscombat	(Right, Number) Returns non-zero if combat mode is currently enabled within the Tactical Console. Example: result = state.iscombat
isendturn	(Right, Number) Returns non-zero if all combatants have taken their allotted actions and it is valid to end the current combat turn. Example: result = state.isendturn
combatturn	(Right, Number) Returns the current combat turn that is underway within the Tactical Console. Example: result = state.combatturn

initchange	(Right, Number) Returns non-zero if any actor has a user-modified initiative value. Example: result = state.initchange
actorcount	(Right, Number) Returns the total number of actors within the portfolio, including all minions. Example: result = state.actorcount
reload[*table*]	(Right, Number) Forces a re-load and re-sort of the table portal whose id is given by the *table* parameter. This target reference is only accessible from within a Trigger Script. Example: result = state.reload[myportal]
value[*id*]	(Left, Right, Number) Provides direct access (read/write) to a global value with the given *id*. This value is globally defined, outside the scope of any actors, and it is persistent. The state of all global values is saved with the portfolio and restored on a reload. A global value must be set before it is retrieved, else a run-time error is reported. Example: result = state.value[id] **NOTE!** This mechanism makes it possible to manage state across the entire portfolio.
setrandom[*id*,*cnt*]	(Right, Number) Creates a new set of random values with the given *id*. The set contains the integer values zero through *cnt*-1, with the values being in a random sequence. The set of values is globally defined, outside the scope of any actors, and it is persistent. The state of the set will be saved with the portfolio and restored on a reload. The value returned is always zero. Example: result = state.setrandom[setid,42] **NOTE!** This mechanism makes it possible to simulate set-based behaviors, such as a deck of cards. The other target references beginning with the "set" prefix below are used in conjunction with this set.
setextract[*id*]	(Right, Number) Extracts and returns the next value from the set with the given *id*, which must already be created. If there are no values left within the set, a run-time error is reported and the value zero is returned. Example: result = state.setextract[setid]
setremain[*id*]	(Right, Number) Returns the number of values that remain within the set with the given *id*. Example: result = state.setremain[setid]
setdiscard[*id*,*val*]	(Right, Number) Locates the value *val* within the set with the given *id* and discards it from the set. Once discarded, the set behaves as if the value no longer exists within the set. Example: result = state.setdiscard[setid,42]

Category: Kit Reference

Context Transitions

Context: HL Kit ... Kit Reference

Contents

Overview

Every script begins with an initial context that is dictated by the particular script. Quite often, though, you'll want to access information somewhere else within the data hierarchy. That's when context transitions come into play. A context transition allows you to move through the hierarchy, progressing to objects either above or below the current context. These transitions can be chained, allowing you to move through a sequence of contexts to reach the desired destination.

NOTE! If transitions are utilized that result in an invalid (i.e. non-existent) context, any subsequent target reference will be invalid. If this occurs during run-time, the operation will be ignored and the target identifier will return zero. A suitable error will generally be displayed, but not always. An example of an invalid context is when a pick attempts to transition to a field that does not exist within that pick.

Using "this"

Every script has an initial context that is automatically established (see the specific script to know what it is). Normally, this context is implied, so you don't need to do anything to reference that context. However, some authors will want their scripts to clearly indicate when the implied context is being used. To accommodate this, scripts can utilize the reserved word "this" to indicate the implied context.

For example, an Eval Script starts with the pick as its implied context. So you could write a target identifier that checks the validity of that pick as simply "valid". Alternately, you could specify "this" as the context, yielding a target identifier of "this.valid". Either method is perfectly legal and you are welcome to use whichever method you prefer.

NOTE! The "this" reference identifies the implied context only. Therefore, you can only use "this" as the first context reference for a target identifier. If "this" is used anywhere else, a compilation error will occur.

Transitions By Context

From within a given context, you are only able to transition to a specific set of other contexts. The topics below identify what the valid transitions are for each context.

- Container Context Transitions
- Hero Context Transitions
- Pick Context Transitions
- Thing Context Transitions
- Field Context Transitions
- Pool Context Transitions
- Scene Context Transitions
- Layout Context Transitions
- Template Context Transitions
- Portal Context Transitions
- Table Context Transitions
- Value Context Transitions
- State Context Transitions

Special Contexts

Within some scripts, special contexts are supported. These special contexts behave the same way within any script that uses them. However, what they correspond to may be different within each script. The specific scripts where special contexts can be used will explicitly cite the availability of the context in their description. Alternately, some special contexts can be established within a script via certain language mechanisms (e.g. "eachpick"). The behavior of these contexts is outlined below.

eachpick	The "eachpick" context only applies within the context of a "foreach" statement and represents the pick that is currently being iterated upon by the "foreach" statement. The current pick is accessed via the "eachpick." initial context. When used, the script context is switched to the iterated pick, and all subsequent actions are performed relative to that pick.
altpick	The "altpick" context represents an alternate pick that is integral to the script and can be accessed readily. The alternate pick is accessed via the "altpick." initial context. When used, the script context is switched to the alternate pick, and all subsequent actions are performed relative to that pick.
altthing	The "altthing" context represents an alternate thing that is integral to the script and can be accessed readily. The alternate thing is accessed via the "altthing." initial context. When used, the script context is switched to the alternate thing, and all subsequent actions are performed relative to that thing.
focus	The "focus" context only applies when the "setfocus" target reference is utilized on a pick context to establish that pick as a memorized context. The established focus pick context is accessed via the "focus." initial context, after which the script context changes to that pick and all subsequent actions are performed relative to that pick.
actor	The "actor" context only applies when the "setactor" target reference is utilized on a hero context to establish that hero as a memorized context. The established actor context is accessed via the "actor." initial context, after which the script context changes to that actor and all subsequent actions are performed relative to that actor. IMPORTANT! This is a situation where there is a critical distinction between "hero" and "actor". The "actor" context only applies to whatever actor has been established as the "actor focus".
transaction	The "transaction" context represents an alternate pick that is integral to scripts involved in buy and sell transactions of objects. The alternate pick is accessed via the "transaction." initial context. When used, the script context is switched to the alternate pick, and all subsequent actions are performed relative to that pick.

IMPORTANT! All special contexts must be specified at the **start** of an identifier. If not, they will not be acknowledged by the compiler. For example, if the "altpick" special context is supported by a script, the reference "altpick.field[livename].text" would work perfectly. However, the reference "this.altpick.field[livename].text" would fail to compile, since "altpick." is not given as the initial context for the identifier.

Category: Kit Reference

Target References

Context: HL Kit … Kit Reference

Overview

When managing data via scripts, the first step is to establish the correct script context. Once you've identified the desired context, you'll then need to specify the appropriate target reference to retrieve or manipulate the specific information you want. Every script context has an assortment of target references that provide access to data pertaining to that context.

NOTE! If the current script context is invalid, any target reference used within that context. If this occurs during run-time, the operation will be ignored and the target identifier will return zero. A suitable error will generally be displayed, but not always. An example of an invalid context is when a pick attempts to transition to a field that does not exist within that pick.

Target Reference Behaviors

Every target reference is assigned a set of behaviors that dictate how it can be used within scripts. The different behaviors are described in the table below.

Right	This target reference can be utilized on the right side of an assignment statement or with the "perform" statement. The value returned is specified within its description.
Left	This target reference can be utilized on the left side of an assignment statement. It must be assigned a value in accordance with its description.
Number	This target reference utilizes a numeric value, either returning a number or expecting to be assigned a number.
String	This target reference utilizes a text string, either returning a string or expecting to be assigned a string.

Topics

The topics below delineate the various target references that are available within each context.

- Container Target References
- Hero Target References
- Pick Target References
- Thing Target References
- Field Target References
- Pool Target References
- Scene Target References
- Layout Target References
- Template Target References
- Portal Target References
- Value Target References
- Table Target References
- State Target References

Category: Kit Reference

Data Access Examples

Context: HL Kit … Kit Reference

Let's look at a few examples of combining context transitions with target references into a useful target identifier. For the first example, we assume we're writing an Eval Script in which we need to access the calculated bonus conferred by the "strength" ability score for the hero. Since an Eval Script starts out in the context of a pick, we'll just go directly to the hero and drill down to get the value we need. From the hero, we transition to the pick that contains the "strength" ability score. Once we have the pick, we then transition to the "bonus" field. And finally we access the value of that field. In the end, the complete reference might look like the following:

```
hero.child[strength].field[bonus].value
```

What if our hero has a magic sword from which we need to extract information? In this situation, we need to drill down into the child gizmo and then into a pick belonging to that gizmo. Starting from the root hero again, we first access the pick that attaches the gizmo. From there, we transition to the gizmo, after which we can transition to the pick inside the gizmo. Once we have the pick context, we can get the field and access its value. The final result would look something like the following:

```
hero.child[magicsword].gizmo.child[attack].field[bonus].value
```

Now let's assume we have a script on a pick within the above gizmo that needs to access a field on a different pick within the same gizmo. Theoretically, we could bounce all the way out to the hero and drill down, but that approach won't work reliably if we have multiple magic swords on the hero. So the best approach is to move upwards to the container and then back down again into the desired pick. The process would look something like the following:

```
container.child[attack].field[bonus].value
```

The key thing to remember when writing scripts is to always know your initial context. From there, simply identify where you want to reference and then systematically transition, one step at a time. If you take things one step at a time, you'll be able to access the information you need and your scripts will work smoothly.

Category: Kit Reference

Script Types

Context: HL Kit … Kit Reference

The Kit leverages a diverse assortment of scripts for a wide range of purposes. The topics below provide a brief discussion of both the role and behavior of each different type of script. The scripts have been grouped into general categories for improved utility.

Contents

Pick Manipulation

These scripts manipulate the contents of picks during the evaluation cycle.

- Eval Script
- Gear Script

Field Manipulation

These scripts manipulate the contents of fields for both display and constraint.

- Bound Script
- Calculate Script
- Finalize Script
- InitFinalize Script

Validation

These scripts apply validation tests to objects with integrated reporting of errors.

- EvalRule Script
- Validate Script
- Integrity Script

Creation/Deletion

These scripts perform appropriate setup and cleanup of specialized objects.

- Creation Script
- Deletion Script

Visual Positioning

These scripts manage the size and positioning of visual elements within panels and sheets.

- Position Script
- Header Script

Synthesis & Presentation

These scripts synthesize information for display to the user in some fashion, including labels, descriptions, mouse-over information, and stat blocks.

- Description Script
- MouseInfo Script
- Label Script
- TitleBar Script
- HeaderTitle Script
- AddItem Script
- Chosen Script
- LeadSummary Script
- Synthesize Script

Trigger

These scripts are invoked in direct response to user actions, such as merging and splitting stackable gear, controlling combat and turns, etc.

- Trigger Script
- Integrate Script
- NewCombat Script
- EndCombat Script
- NewTurn Script
- Initiative Script
- Merge Script
- Split Script
- Change Script

Transaction

These scripts are associated with the buying and selling of equipment.

- TransactSetup Script
- TransactBuy Script
- TransactSell Script

Mode Transition

These scripts associated with the transition into and out of advancement mode.

- CanAdvance Script
- Transition Script

Release Changes

These scripts are used to accommodate changes between data file releases and potential loading errors of portfolios.

- LoadError Script
- LoadFixup Script

Category: Kit Reference

Eval Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Components, Things
Procedure Use:	"eval" type, "pick" context

The Eval script utilizes the following special symbols:

-None-	There are no special symbols for an Eval script.

Description

The Eval script is the primary workhorse throughout any set of data files. This script is scheduled within the evaluation cycle and where you will orchestrate the virtually all of the effects for the game system. If a special ability confers a bonus to certain skills, you'll use an Eval script to apply them. If attributes confer adjustments to abilities, attacks, damage, or anything else, you'll use Eval scripts to apply them. Calculated abilities, such as attack ratings or casting powers will be determined through Eval scripts. The list is endless.

Every Eval script is associated with a particular thing. If an Eval script is defined for component, then it is inherited by the things which derive from that component. When a thing is added to a container, it becomes a pick, and new instances of every Eval script for that thing are created and managed by HL. You associate all of the primary behaviors with each pick via the Eval scripts defined for the underlying thing.

When invoked, an Eval script starts with its associated pick as its initial context. You are free to navigate through the hierarchy and effect changes anywhere you deem appropriate. However, all of the changes will typically either be made to the associated pick or be made to other objects based on facets of the associated pick.

As a general rule, all of the dynamic effects that are applied throughout your data files should be handled via Eval scripts. Because the timing of all Eval scripts is controlled, you can ensure that all of the different effects of different scripts are applied in a carefully defined sequence. For example, in the d20 System, adjustments to attributes (e.g. from magic items) must be applied before the bonuses conferred by attributes are calculated and applied.

Eval scripts provide a single, generalized mechanism through which you can accomplish just about anything. As a result, the actual behaviors of an Eval script will vary more widely than with any other type of script. If there is one type of script to get comfortable with first, it is definitely the Eval script.

Example

Due to the wide range of behaviors that can be applied via Eval scripts, there is no one good example. The XML below shows three separate Eval scripts that are defined as part of the basic handling of equipment within the Sample data files. Each script is scheduled to be performed at a different time during the overall evaluation cycle, with the timing ensuring that each behavior is properly interleaved with other behaviors triggered by other objects.

```
<!-- All melee weapons get the appropriate tag -->
<eval index="1" phase="Setup" priority="5000"><![CDATA[
  perform assign[Armory.Melee]
  ]]></eval>

<!-- Calculate the net attack roll for the weapon -->
<eval index="2" phase="Final" priority="7000" name="Calc wpNetAtk"><![CDATA[
  field[wpNetAtk].value = #trait[skMelee] + field[wpBonus].value + field[wpPenalty].value
  ]]></eval>

<!-- Prepend any derived special notes to the appropriate field -->
<eval index="3" phase="Render" priority="2000"><![CDATA[
  var special as string

  ~assign any appropriate special notes to the "special" variable here

  ~prepend any existing special details with the notes for this weapon
  if (empty(special) = 0) then
    if (field[wpNotes].isempty = 0) then
```

```
       special &= ", "
       endif
     field[wpNotes].text = special & field[wpNotes].text
     endif
   ]]></eval>
```

Category: Kit Reference

Gear Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Things
Procedure Use:	None

The Eval script utilizes the following special symbols:

-None-	There are no special symbols for a Gear script.

Description

The purpose of the Gear script is to let you apply non-standard behaviors to gear. By default, all gear that is placed within a holder has its weight accrued into the net weight of the holder. For example, if a backpack normally weighs 3 pounds and there is a potion inside the backpack that weighs 2 pounds, the net weight of the backpack will be 5 pounds. However, there are some items that break this rule. The classic example is the Bag of Holding from the d20 System, which weighs a fixed amount, regardless of what is stored within it. The Gear script allows you to override the standard behaviors and assign a custom weight to the holder.

The Gear script begins with an initial context of the pick itself. In the example above, the Gear script would start with the Bag of Holding as its initial context. From there, you can access other facets of the structural hierarchy if you need them. Upon entry to the script, the "gearHeld" field value has been automatically calculated to be the total weight of all items held within. In addition, the "gearNet" field value contains the net weight of the item, including the weight of both the item itself and its contents. By setting the values of these fields appropriately in the script, the appropriate weights will propagate up the containment hierarchy.

Example

We'll continue with the example of the Bag of Holding here. For this item, you would simply set the weight of the pick to be its basic weight, ignoring the accrued weight of its contents.

```
~Our contents count for no weight at all!
field[gearNet].value = field[gearWeight].value
```

Category: Kit Reference

Bound Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Field
Procedure Use:	"bounds" type, "pick" context

The Bound script utilizes the following special symbols:

minimum	(Number) Entry: The default minimum value to be used for bounding. Exit: The final minimum value to be used for bounding.
maximum	(Number) Entry: The default maximum value to be used for bounding. Exit: The final maximum value to be used for bounding.

Description

The Bound script allows you to dynamically calculate the appropriate minimum and maximum values that a field can possess. This is invaluable when the minimums are dependent upon dynamically changing criteria. For example, consider the height and weight of a character. If the game system supports different races, then each race will have its own minimums and maximums for these characteristics. You need to be able to set the bounds based on the race that has been selected.

The Bound script begins with an initial context of the pick containing the field being bounded. This makes it possible to easily access other fields on the pick to base the bounding limits. The most common use of this script is for the containing pick to have separate fields that dictate the minimum and maximum values to be imposed. These fields are properly setup by Eval scripts and then the Bound script simply assigns those values as the limits.

When the Bound script is invoked, the minimum and maximum limits start out as the limits specified via the "minimum" and "maximum" attributes on the field definition. If no limits are defined via those attributes, then there is no limit on the field value. This makes it possible to have your Bound script use the defaults and only impose appropriate limits when necessary, leaving the limits unchanged otherwise.

Until a Bound script is invoked, the value of the field is bounded against the default limits specified by the field attributes. So any use of the field value within a script that occurs earlier than the Bound script will be restricted based on those limits. Once the Bound script is invoked, the current field value is immediately bounding to the new limits. Thereafter, any changes applied to the field value are automatically bounded to the new limits.

Example

The bounding of age, height, and weight uses the approach outlined above, where separate fields are used to dictate the appropriate bounding limits. As such, the Bound script for the character's age simply pulls its limits from the separate fields, as shown below.

```
@minimum = field[perAgeMin].value
@maximum = field[perAgeMax].value
```

Category: Kit Reference

Calculate Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Fields
Procedure Use:	"calculate" type, "pick" context

The Bound script utilizes the following special symbols:

value	(Number) Entry: The current value of the field (if the field is numeric), else zero. Exit: The final value to be used for the field (if the field is numeric).
text	(String) Entry: The current context of the field (if the field is text), else the empty string. Exit: The final contents to be used for the field (if the field is text).

Description

The Calculate script is in many ways a specialized Eval script, with the explicit purpose of calculating the value of a field. The script is scheduled to occur at some point during the evaluation cycle, and it could just as easily be implemented as an Eval script. The advantage of the Calculate script is purely a semantic organizational benefit, since the script is defined directly on the field instead of generally on the component (as an Eval script would be handled).

The Calculate script allows you to calculate the appropriate contents for a field. Since the script is scheduled during evaluation, other scripts can legally modify the field before or after the script. Consequently, it is usually only appropriate to use a Calculate script for fields that can be calculated in one centralized location (this script). Otherwise, it is typically better to use Eval scripts for all manipulations of the field.

The Calculate script begins with an initial context of the pick containing the field being calculated. This makes it possible to easily access other fields on the pick when calculating the new contents. When invoked, the appropriate special symbol is initialized with the current contents of the field. If the field is numeric, the "value" special symbol contains the value, and the "text" special symbol contains the current contents of a text field. When the script completes, the updated contents of the appropriate special symbol are used for the field, with the other special symbol completely ignored.

Example

Within the Sample data files, the final value of a trait is determined via a Calculate script. The field value is the result of adding the user-assigned value, any bonuses assigned via scripts, and any in-play adjustment.

```
@value = field[trtUser].value + field[trtBonus].value + field[trtInPlay].value
```

Category: Kit Reference

Finalize Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Fields
Procedure Use:	"finalize" type, "pick" context

The Finalize script utilizes the following special symbols:

value	(Number) Entry: The current value of the field (if the field is numeric), else zero. Exit: Ignored.
text	(String) Entry: The current contents of the field (if the field is text), else the field value converted to a string. Exit: The contents to be used for finalized version of the field. The text may contain encoding.
ispick	(Number) Entry: Indicates whether the value is being finalized for a pick (non-zero) or a thing (zero). Exit: Ignored.

Description

The role of the Finalize script is to synthesize a text version of a numeric field that consists of more than just the field value. By default, a numeric field is automatically converted to a string when the ".text" target reference is used on the field, so this script is of no use unless you want a different behavior.

There are numerous places where a Finalize script can be extremely handy. Lots of fields are managed as numeric values for easy of manipulation in data files. However, many of those fields need to be displayed to the user with appropriate units annotated. For example, the cash possessed by a character and the costs of equipment are tracked as numeric values but should ideally be displayed as currency (e.g. "$142"). The height of a character may be tracked as inches, but the user wants to view it in a more common form (e.g. 5'11"). The Finalize script makes it easy to accomplish this.

The Finalize script is invoked after the entire evaluation cycle has completed, but it is only invoked when something actually asks for the finalized contents of the field. Once the script is invoked once, the results are cached internally, but the cost of invoking the script is not incurred unless the contents are actually used somewhere.

The Finalize script begins with an initial context of the pick containing the field being calculated. When invoked, the "value" special symbol is assigned the actual value of the field at the end of evaluation. If the field is text, then the value is always zero. Similarly, the "text" special symbol is assigned the text-based contents of the field after evaluation. If the field is numeric, the value is automatically converted to the corresponding string. When the script completes, the updated contents of the "text" special symbol are used as the finalized contents for the field.

NOTE! When accessing a numeric field via a script, you have the option to use either the ".value" or ".text" target reference. Within any script that utilizes finalized field values (indicated at the top of each script), you should always use the ".text" target reference, unless you have a specific reason for using the ".value" target reference. The reason for this is that ".text" will always retrieved any finalized version of the field, while ".value" will always retrieve the actual value, without any finalization logic being applied.

Example

Within the Sample data files, the height of a character is managed as a numeric field that handles everything in terms of inches. The user wants to view this in the standard format using feet and inches (e.g. 5'11"). A Finalize script is defined that accomplishes this, as shown below.

```
~calculate the height in terms of feet and inches
var feet as number
var inches as number
feet = @value / 12
feet = round(feet,0,-1)
inches = @value - (feet * 12)

~synthesize appropriate text to display the height properly
@text = feet & "'"
```

```
if (inches <> 0) then
  @text = @text & " " & inches & chr(34)
  endif
```

Category: Kit Reference

InitFinalize Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Global Behavior
Procedure Use:	"finalize" type, "pick" context

The InitFinalize script utilizes the following special symbols:

value	(Number) Entry: The current value of the initiative determined for the actor. Exit: Ignored.
text	(String) Entry: The current initiative value converted to a string. Exit: The contents to be used for finalized version of the field. The text may contain encoding.

Description

The InitFinalize script is a special-purpose instance of a normal Finalize script. The value being finalized is the initiative value calculated for an actor via the Initiative script. In all respects, this script behaves as a standard Finalize script, except that there are no other field values that can be accessed via the initial pick context.

Example

This is a standard Finalize script, so the example for that script can be referenced.

Category: Kit Reference

EvalRule Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Components, Things
Procedure Use:	"evalrule" type, "pick" context

The EvalRule script utilizes the following special symbols:

valid	(Number) Entry: Assumed the rule is **not** satisfied with a value of zero. Exit: Indicates whether the rule is satisfied (non-zero) or not satisfied (zero).
message	(String) Entry: Contains the default message text if the rule is not satisfied. Exit: Contains the final message text to display if the rule is not satisfied. The text may contain encoding.
summary	(String) Entry: Contains the default summary text if the rule is not satisfied. Exit: Contains the final summary text to display if the rule is not satisfied. If not specified, but the "message" symbol is modified, the summary automatically uses the message text.

Description

The EvalRule script is very similar to an Eval script, with the primary distinction being that the EvalRule script verifies a rule is observed instead of applying changes. This script is scheduled during the evaluation cycle, so it allows you to check the state of an actor at intermediate points, in addition to at the end of evaluation. Intermediate checks make it possible to verify values before they are further modified by other aspects of the character.

An EvalRule script shares all the same behaviors of an Eval script, plus it adds the rule to be validated. The rule can be anything you want. However, the rule is always either satisfied or not satisfied. This is determined by the "valid" special symbol. The symbol starts out as zero, indicating that the rule is not satisfied. At any point during the script, you can set the symbol to non-zero, thereby indicating that the rule **is** satisfied. You can also change the symbol value throughout the script if you want. The only value that matters is the final value when the script ends, where a non-zero value indicates the rule is satisfied.

If the validation test is not satisfied, the message and summary for the rule are displayed within the validation report and validation summary for the actor, respectively. Since the message and summary are typically static requirements (e.g. an attribute must be a value of X or more), you will rarely need to modify them. However, there will be cases where you want to tailor the message and/or summary to provide more detailed information based on dynamic information. When these situations arise, simply set the "message" and/or "summary" special values to the desired contents and they will be used.

Since an EvalRule script is equivalent to an Eval script, it **is** possible to apply changes throughout the structural hierarchy via the script. However, it is generally a bad idea to do that, since the purpose of an EvalRule script is to validate a rule - not effect change. We therefore recommend that you avoid applying changes within EvalRule scripts.

Like an Eval script, every EvalRule script is associated with a particular thing. If an EvalRule script is defined for a component, then it is inherited by the things which derive from that component. When a thing is added to a container, it becomes a pick, and new instances of every EvalRule script for that thing are created and managed by HL. You associate all of the validation rules with each pick via the EvalRule scripts defined for the underlying thing.

When invoked, an Eval script starts with its associated pick as its initial context. You are free to navigate through the hierarchy and test the state of anything that impacts the rule. However, a given EvalRule script should typically either pertain to the associated pick or pertain to the actor as a whole.

Example

An EvalRule script can be used to test virtually anything. Within the Sample data files, a rule is used to ensure that pieces of gear are not both equipped and held within a container (e.g. a sword being equipped and stored in a backpack). The rule below will verify this condition.

```
~if not both equipped and held within a container, we're valid
if (field[grIsEquip].value + isgearheld < 2) then
```

```
    @valid = 1
    done
    endif

~mark the tab as invalid
linkvalid = 0
```

Category: Kit Reference

Validate Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Container
Alternate Context:	Pick or Thing
Fields Finalized?	Yes
Where Used:	Components, Things
Procedure Use:	"validate" type, "pick" context

The Validate script utilizes the following special symbols:

valid	(Number) Entry: Assumes the pre-requisite is **not** satisfied with a value of zero. Exit: Indicates whether the pre-requisite is satisfied (non-zero) or not satisfied (zero).
message	(String) Entry: Contains the default message text if the pre-requisite is not satisfied. Exit: Contains the final message text to display if the pre-requisite is not satisfied. The text may contain encoding.
ispick	(Number) Entry: Indicates whether the pre-requisite is being applied to a pick (non-zero) or thing (zero). Exit: Ignored.

Description

The Validate script is exclusively used within pre-requisite tests. The script verifies that a specific pick or thing satisfies a particular pre-requisite relative to its container. If the requirement is not satisfied, then the object is designated as invalid. If the object is a pick that has already been added to the portfolio, the specified message is displayed within the validation report for the actor.

The pre-requisite test can be anything you want, provided only the object and the container are considered by the test. The requirement is always either satisfied or not satisfied. This is determined by the "valid" special symbol. The symbol starts out as zero, indicating that the requirement is not satisfied. At any point during the script, you can set the symbol to non-zero, thereby indicating that the requirement **is** satisfied. The only value that matters is the final value when the script ends, where a non-zero value indicates the pre-requisite is satisfied.

Each pre-requisite is associated with a particular thing. If a pre-requisite is defined for a component, then it is inherited by the things which derive from that component. The pre-requisites are checked for every thing when it is presented as an option for the user to add via a table or chooser. If one or more pre-requisites are not satisfied, the thing is designated as invalid and the failed requirements are shown in the description information for the thing. If the thing is added by the user in spite of the failed pre-requisites, then all the pre-requisites are applied to the pick at the end of every evaluation cycle.

When invoked, a Validate script starts with the container as its initial context. If the object is a pick, the container is the one the pick resides within. If the object is a thing, the container is the prospective container to which the thing will potentially be added. The reason for the container as the initial context is that the pre-requisite test is defined on the thing. Consequently, the thing will know about itself and will typically be looking to verify that the prospective container satisfies the needs of the thing.

For those situations where you also need to access characteristics of the pick or thing, the Validate script provides an alternate context. This context will always be the pick or thing, as appropriate. You can use the "ispick" special symbol to determine whether the pre-requisite is being applied to a pick or a thing. After that, you can use the "altpick" or "altthing" context to access the object.

NOTE! The use of pre-requisites with a Validate script is generally only needed in more complex situations. The "pickreq" and "exprreq" mechanisms are much simpler to use and maintain, and they should cover 90% of the situations where you need to establish a pre-requisite.

Example

A simple pre-requisite might establish a dependency on an attribute value being at least equal to some number. The example below shows a Validate script that tests whether the container has a child pick (strength) that has a final value of at least 13.

```
if (child[attrStr].field[trtFinal].value >= 13) then
  @valid = 1
  endif
```

Category: Kit Reference

Integrity Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Container
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Entities
Procedure Use:	"container" context

The Integrity script utilizes the following special symbols:

message	(String) Entry: Contains the empty string to indicate that no errors were identified. Exit: Contains the final error message text to display. If there are no errors, specify the empty string. The text may contain encoding.

Description

The Integrity script is associated with entities. This script allows the author to verify that all the appropriate values have been specified for a gizmo before allowing the user to actually save changes to the gizmo.

The Integrity script is invoked when the user finishes editing a gizmo derived from the entity for which the script is defined. When the user tries to save his changes, the script is triggered, at which point the script verifies that an assortment of characteristics are satisfied by the gizmo. If any are not satisfied, an error message is displayed to the user and the user is not allowed to save his changes until the errors are corrected.

It is the responsibility of the script to synthesize the message shown to the user that reports the errors in the gizmo. If an error is detected in the script, a suitable error message should be appended to the "message" special symbol. If multiple errors occur, the message should contain a list of them, with each message on new line. If there are no errors, then the special symbol should be the empty string, and that tells HL that the gizmo is safe to save.

When invoked, an Integrity script starts with the gizmo as its initial context. There is generally no reason to navigate outside the context of the gizmo, but it is technically allowed. The primary focus will be on the gizmo itself and the child picks within it.

NOTE! Don't be too stringent with the Integrity script. Remember that every gaming group has its own house rules that will modify the basic game rules. As such, it's important to use validation rules to trap most errors instead of requiring the user to obey certain rules. The Integrity script should only be used to enforce details that should always be required and/or that impact your ability to write quality data files.

Example

The handling of advancement that is found within the Sample data files utilizes an Integrity script. Based on the information shown for a particular advancement, certain fields must be specified by the user. If that information is not provided, the form remains shown and no changes to the gizmo are saved.

```
~setup a bullet character we can put at the start of each error
var bullet as string
bullet = "{bmp bullet_red}{horz 4}"

~verify that a chooser selection is made if one is required
if (parent.tagis[Advance.MustChoose] <> 0) then
  if (firstchild["Advance.Gizmo"].tagis[Advance.Gizmo] = 0) then
    @message = @message & bullet & "A specific trait/ability must be selected via the chooser.\n"
    endif
  endif

~verify that we've been given a domain if one is required
if (parent.tagis[Advance.MustChoose] + parent.tagis[Advance.AddNew] >= 2) then
  if (firstchild["Advance.Gizmo"].tagis[User.NeedDomain] <> 0) then
    if (empty(child[advDetails].field[advUser].text) <> 0) then
      @message = @message & bullet & "The selection requires that you specify an appropriate
domain.\n"
      endif
    endif
  endif
```

Category: Kit Reference

Creation Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Components
Procedure Use:	None

The Integrity script utilizes the following special symbols:

-None-	There are no special symbols for a Creation script.

Description

The Creation script is defined for components, and it pertains to all picks that derived from a given component. The script is invoked a single time when a pick is first created. As such, its primary use is for performing special setup and/or configuration of picks.

For example, in the d20 System, there is an optional rule where characters always receive maximum hit points for each new level. This can be handled quite easily via a Creation script. If the user has enabled the setting, the hit points are set to the maximum, else they are set to zero so that the user needs to modify them appropriately.

When invoked, a Creation script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

Example

Using the d20 System example cited above, the Creation script below will properly initialize the hit points field for a new class level based on the configuration setting.

```
if (hero.tagis[source.MaxHP] <> 0) then
  field[lvlHP].value = field[lvlHitDice].value
else
  field[lvlHP].value = 0
  endif
```

Category: Kit Reference

Deletion Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Components
Procedure Use:	None

The Integrity script utilizes the following special symbols:

-None-	There are no special symbols for a Deletion script.

Description

The Deletion script is the analog of the Creation script. The script pertains to all picks that are derived from a given component, and it is invoked a single time just before a pick is finally destroyed. As such, its primary use is for performing special wrapup handling of picks.

This script is rarely used, but it is quite necessary in those places. The easiest example is with usage pools, such as those used by the journal mechanism. Each journal pick adds its awards (e.g. XP, cash) to the overall usage pools maintained for the actor. But if the user deletes a journal pick, it's critical that the awards conferred by the pick be subtracted back out. Enter the Deletion script, which serves perfectly for this purpose.

When invoked, a Deletion script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

Example

Using the example cited above, the Deletion script below will properly subtract out the adjustments from a journal pick before it is destroyed.

```
perform hero.usagepool[TotalXP].adjust[-usagepool[JrnlXP].value]
perform hero.usagepool[TotalCash].adjust[-usagepool[JrnlCash].value]
```

Category: Kit Reference

Position Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Scene or Layout or Template
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Templates, Layouts, Panels, Forms, Sheets
Procedure Use:	None

The Position script utilizes the following special symbols:

-None-	There are no special symbols for a Position script

Description

The location of visual elements on the display is controlled via the Position script. This script is used the same way within scenes, layouts, and templates. Through this script, the visual elements contained within are sized and positioned. For example, a Position script for a template will coordinate the sizing and positioning of the various portals that are defined within the template.

Most visual elements are sized and positioned by their containing element. For example, a layout is often told by the containing scene how much space it gets to utilize and then it sizes it contents to fit within that space. However, there are times when visual elements must size themselves, so it is also possible for a visual element to size itself within its Position script. An example of this is where you have a template within a table that needs to determine its own height based on the information being shown within.

Within a visual container, the position of elements is always relative to the upper left corner of the container. If the container has a margin assigned, then upper left corner is adjusted accordingly. This ensures that the visual container can be positioned anywhere within its container, and the visual elements within don't need any knowledge of the container's position.

IMPORTANT! The Position script for a template within a table is invoked separately for each item shown within the table. If the template is within a table containing items of non-varying height (the norm), the Position script is also invoked one **additional** time. The occurs **before** each item is processed and serves the purposes of calculating the of each item. When invoked for sizing, all text-based fields are empty and all value-based fields have a value of zero. The "issizing" target reference allows detection of this special use.

NOTE! The Position script is read-only with respect to the contents of the portfolio, although visual elements may be modified. Within this script, all aspects of the hierarchy can be accessed, but nothing can be changed.

Example

Assume we have a template that contains two portals: a label portal and an edit portal. The template is used to let the user edit the name of something, so the label portal displays "Name:" and the edit portal allows the user to edit the name. The label portal will be automatically sized to the width of its text, so all we need to do is size the edit portal and position both portals. The Position script for a simple template like this might look like the following.

```
~center both portals vertically
perform portal[label].centervert
perform portal[edit].centervert

~pick a width for the edit portal
portal[edit].width = 150

~put the label on the left and the edit portal on its right
portal[label].left = 0
perform portal[edit].alignrel[ltor,label,10]
```

Category: Kit Reference

Header Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Template
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Templates
Procedure Use:	None

The Header script utilizes the following special symbols:

-None-	There are no special symbols for a Header script

Description

The Header script behaves very similarly to the Position script. This script's purpose is to handle the sizing and positioning of visual elements in one specific situation. When the same template is used within a table for positioning both items **and** a header, the Header script serves to position the header portals. If two separate templates are used for items versus header, the Position script for each is used and the Header script is not employed.

Although the behaviors are similar, the Header script is distinct from the Position script in a variety of ways. First of all, portals within the template that are designated as "isheader" are only accessible via the Header script, hence the name. Second, the Header script is invoked **after** the normal Position script. This allows the header contents to be sized and positioned relative to the final locations for non-header portals. To make this easy, the Header script **can** access **both** header and non-header portals, although any positioning changes to non-header portals is ignored.

For more details, please see the section on dual-purpose headers.

Example

We'll assume that there are two fields within each item of the table that we want to put a simple header above. Let's call them "damage" and "range", and each is shown within its own portal. These two portals are positioned via the Position script, so the Header script will position the header labels directly above those portals.

```
~our header height is the height of our labels
height = portal[hdrdamage].height

~if this is a "sizing" calculation, we're done
if (issizing <> 0) then
  done
  endif

~each of our header labels has the same width as the corresponding data beneath
portal[hdrdamage].width = portal[damage].width
portal[hdrrange].width = portal[range].width

~center each header label on the corresponding data beneath
perform portal[hdrdamage].centeron[horz,damage]
perform portal[hdrrange].centeron[horz,range]

~align all header labels at the bottom of the header region
perform portal[hdrdamage].alignedge[bottom,0]
perform portal[hdrrange].alignedge[bottom,0]
```

Category: Kit Reference

Description Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Pick or Thing
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"description" type, "info" context, "pick" context

The Description script utilizes the following special symbols:

text	(String) Entry: Contains the default description for the object, which consists of the basic description text assigned to the object and any failed pre-requisites. Exit: Contains the text to be displayed to the user as the description. The final text may contain encoding.
ispick	(Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

Description

Whenever the user is presented with a selection list to choose from, the list of available items is shown on the left and a description area is provided on the right. The description area contains all the details for the currently highlighted item. By default, this consists of the basic description text assigned to the item and any pre-requisites failed by the item. However, the author can present more detailed information via the use of a Description script. This allows the author to factor in context-driven material, such as pertinent field values and adjusted values due to the influence of previous selections.

NOTE! The Description script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

When displaying weapons, it's useful to show the damage and range. This can be easily appended to the default text for display.

```
@text &= "\n"
@text &= "Damage: " & field[wpDamage}.text & "\n"
@text &= "Range: " & field[wpRange].text & "\n"
```

Category: Kit Reference

MouseInfo Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick or Thing
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"mouseinfo" type, "info" context, "pick" context

The MouseInfo script utilizes the following special symbols:

text	(String) Entry: Contains the default mouse-info for the object, which consists of the basic description text assigned to the object and any failed pre-requisites. Exit: Contains the text to be displayed to the user as the mouse-info text. The final text may contain encoding.
ispick	(Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

Description

HL makes extensive use of visual elements that the user can move the mouse over to obtain further details about an object. This is generally referred to as mouse-over highlighting, or simply *mouse info*, and it is accomplished via the MouseInfo script on various portals. The purpose of the MouseInfo script is to synthesize the actual text to be displayed for the pick/thing that the user is inquiring about.

NOTE! The MouseInfo script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

When displaying weapons, it's useful to show the damage and range. This can be easily appended to the default text for display.

```
@text &= "\n"
@text &= "Damage: " & field[wpDamage}.text & "\n"
@text &= "Range: " & field[wpRange].text & "\n"
```

Category: Kit Reference

Label Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick or Thing
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"label" type, "info" context, "pick" context

The Label script utilizes the following special symbols:

text	(String) Entry: Contains the pre-defined text for the label portal. Exit: Contains the text to be displayed to the user as the label portal contents. The final text may contain encoding.
ispick	(Number) Entry: Indicates whether the text is being rendered for a thing (zero) or a pick (non-zero), allowing different handling to be performed for the two separate cases. Exit: Ignored.

Description

The Label script is used exclusively within label portals, and it is the script defined for just-in-time rendering of the portal contents. Instead of pulling the information for display out of a field or using literal text, a script is defined. This script is invoked every time the display updates, allowing the script to re-calculate the information to be displayed based on the most recent state of the portfolio.

NOTE! The Label script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

IMPORTANT! Avoid using Label scripts indiscriminately. Since the script must be re-evaluated and re-displayed with every update of the interface, they can be computationally expensive and slow down performance on slower computers. In many cases, a separate field can be used in which the appropriate value can be calculated and placed. When a field is used, HL will only update the display if the value actually changes.

Example

Within the d20 System data files, the hero's HP and AC are both shown at the top. Separate label portals could be used for each, but the width of the HP will vary from one to three digits, requiring scripts to re-calculate the positions of the portals all the time in case the number of digits has changed. It can be more efficient to simply use a single Label script that synthesizes the text to be displayed and eliminates all the re-positioning, resulting in a script like the one below.

```
@text = "HP: " & herofield[acHP].text & " AC: " & herofield[acArmorCls].text
```

Category: Kit Reference

TitleBar Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Container
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"titlebar" type, "entity" context, "container" context

The TitleBar script utilizes the following special symbols:

text	(String) Entry: Contains the default text to display ("Choose an Item from the List Below"). Exit: Contains the text to be displayed to the user to prompt selection. The final text may contain encoding.

Description

The TitleBar script is only used within portals that display a choose form where the user can select an item. At the top of the choose form is a title bar that contains a prompt, directing the user to select an item from the list presented. This script allows an author to specify the exact text to be displayed at the top of the selection table, making it possible to remind the user of the context being selected or how many selections remain.

NOTE! The TitleBar script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

When displaying selections where the user can select multiple items, it's helpful to inform the user how many choices remain. This can be easily achieved by a script like the one below.

```
@text = "Add a Special Ablity - " & hero.child[resAbility].field[resSummary].text
```

Category: Kit Reference

HeaderTitle Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"label" type, "info" context, "pick" context

The HeaderTitle script utilizes the following special symbols:

text	(String) Entry: Contains the empty string. Exit: Contains the text to be displayed to the user as the title above the table. The final text may contain encoding.

Description

The HeaderTitle script is essentially a special-purpose Label script used in conjunction with tables. This script allows you to synthesize the text to be displayed as a simple header for a table, without all the work of defining a custom template. It display a single line of text, centered within the header region above the table.

The initial context for the script is dictated by the "headerpick" attribute for the table. The engine will locate a pick within the container that is derived from the specified thing, and that will be the starting script context. Aside from that, the HeaderTitle script is a standard Label script.

When used for tables shown on the screen, the header text is centered within the header region and defaults to using a 10-point Arial font in soft white with a bold style. When used within sheet output, the text is centered in the header region with a solid, light-grey background, and it uses a 13-point Arial font in black with a bold style.

NOTE! The HeaderTitle script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

The Skeleton files make extensive use of this mechanism for putting a suitable title above each table. For example, above the table for selecting Special Abilities, the header shows an appropriate title that includes how many selections remain.

```
@text = "Special Ablities: " & hero.child[resAbility].field[resSummary].text
```

Category: Kit Reference

AddItem Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"label" type, "info" context, "pick" context

The AddItem script utilizes the following special symbols:

text	(String) Entry: Contains the empty string. Exit: Contains the text to be displayed to the user as the "add" item at the bottom of the table. The final text may contain encoding.

Description

The AddItem script is essentially a special-purpose Label script used in conjunction with tables. This script allows you to synthesize the text to be displayed within a simple "add item" at the bottom of a table, without all the work of defining a custom template. The "add item" for a table enables the user to click within it to add a new item to the table.

The initial context for the script is dictated by the "addpick" attribute for the table. The engine will locate a pick within the container that is derived from the specified thing, and that will be the starting script context. Aside from that, the AddItem script is a standard Label script.

The simple "add item" provided via the mechanism displays a single line of text, centered within the item at the bottom of the table. The text is centered within the item and defaults to using a 10-point Arial font in soft white with a bold style.

NOTE! The AddItem script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

The Skeleton files make extensive use of this mechanism for putting a tailored item at the bottom of each table that prompts the user to add the appropriate kind of object. You can also color-code the text to indicate whether items need to be added or too many items have been added. For example, at the bottom of the table for selecting Special Abilities, the "add item" shows an appropriate message and highlights it based on how many selections remain.

```
~set the color based on whether the proper number of slots are allocated
if (field[resLeft].value = 0) then
  @text = "{text a0a0a0}"
elseif (field[resLeft].value < 0) then
  @text = "{text ff0000}"
  endif
@text &= "Add New Special Ability"
```

Category: Kit Reference

Chosen Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	None

The Chosen script utilizes the following special symbols:

text	(String) Entry: Contains the default text to display ("-Please Select-"). Exit: Contains the text to be displayed to the user as the chosen selection. The final text may contain encoding.
ispick	(Number) Entry: Indicates whether the chooser contains a selection or not. Exit: Ignored.

Description

The Chosen script is essentially a special-purpose Label script used in conjunction with choosers. This script allows you to synthesize the text to be displayed as the current selection within a chooser. If nothing has been selected yet, you can prompt the user to do so, and you can use color-coding to indicate errors with the current selection.

The initial context for the script is the currently selected item. If nothing is selected, the initial context with be invalid, so you're limited to displaying something simple. Aside from that, the Chosen script is a standard Label script.

NOTE! The Chosen script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

Choosers provide an excellent solution when the user must choose exactly one option from an assortment. An excellent example is race, where a character must be assigned a single race. The code below shows the Chosen script for the race chooser within the Skeleton files, including color highlighting when no selection has yet been made.

```
if (@ispick = 0) then
  @text = "{text ff0000}Select Race"
else
  @text = "Race: " & field[name].text
  endif
```

Category: Kit Reference

LeadSummary Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Hero
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Definition File
Procedure Use:	"container" context

The LeadSummary script utilizes the following special symbols:

text	(String) Entry: Contains the text "No Summary Provided". Exit: Contains the text to be output as the summary for the lead actor. The final text may contain encoding.

Description

When a user imports characters from a saved portfolio, a summary is shown next to the name. This summary provides basic details about the character that are unique to each game system. The LeadSummary script is used to synthesize the summary. The script is invoked whenever a portfolio is saved so that the summary can be stored in the portfolio and subsequently retrieved for display during import.

The LeadSummary script starts with the leading actor as its initial context. You can then cull whatever information you need from the actor for inclusion within the summary.

NOTE! The LeadSummary script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

Example

In the Mutants & Masterminds game system, each character is shown with its current power points and power level. The script below shows how this is done.

```
~Show our total PP and PL
@text = herofield[SpentPP].value & " PP"
@text &= ", PL " & herofield[CurrentPL].value
```

Category: Kit Reference

Synthesize Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Hero
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Dossiers
Procedure Use:	"synthesize" type, "container" context

The Synthesize script utilizes the following special symbols:

newline	(String) Entry: Contains the appropriate text to begin a new line of text within the output being synthesized (i.e. insert a line break). Exit: Ignored.
boldon	(String) Entry: Contains the appropriate text to turn on output of bold text. Exit: Ignored.
boldoff	(String) Entry: Contains the appropriate text to turn off output of bold text. Exit: Ignored.
italicson	(String) Entry: Contains the appropriate text to turn on output of italic text. Exit: Ignored.
italicsoff	(String) Entry: Contains the appropriate text to turn off output of italic text. Exit: Ignored.
separator	(String) Entry: Contains the appropriate text to insert a suitable horizontal separator. In HTML output, this is the "<hr>" element, and elsewhere it's a line consisting of a string of dashes. Exit: Ignored.

Description

The Synthesize script is utilized when generating text output for a character dossier. A classic example is the creation of statblock output. When the user requests text output, he will also specify the style to be used when synthesizing the output (e.g. HTML, BBCode, or plain text). Before the script is invoked, HL will setup a number of appropriate mechanisms for that output style, which are provided via the various special symbols. For example, in HTML output, the "@boldon" special symbol will map to "<b>", while it will be "[b]" for BBCode output and do nothing for plain text output. This makes it possible for you to write a single Synthesize script that will generate output properly for each different style, without having to do any special handling within the script.

Unlike other scripts, the Synthesize script does not use a "@text" special symbol. Since the volume of output in some cases will be significant, using the standard approach simply isn't practical or efficient. Instead, the output is generated in sequential fashion, and the "append" language statement is used to accomplish this (see details). The "append" statement allows you to systematically construct the output, one section at a time. Once something is output, you can forget about it and let HL handle it.

If you need to do special formatting within a Synthesize script that is based on the output style, you can. You can find out the style by using the the "isdossierstyle" script target reference from within the State script context.

The Synthesize script starts with the actor to be output as its initial context. You can then cull whatever information you need from the actor for inclusion within the output.

NOTE! The Synthesize script is read-only. Within this script, virtually all aspects of the structural hierarchy can be accessed, but nothing can be changed.

NOTE! While within a Synthesize script and any procedures that are invoked, a global tag is automatically assigned by HL. The tag belongs to the "dossier" tag group and has the unique id of the dossier being output. This allows the script code to base its behavior on the actual dossier that the user is outputting.

Example

Every statblock starts with the character's name and then continues with other appropriate information that depends on the game system. The example below shows the start of a Synthesize script that includes the name, race, and age of the character.

```
var txt as string

~start by getting our name
if (empty(hero.actorname) = 0) then
  txt = hero.actorname
else
  txt = "Unnamed Character"
  endif

~output our name
append @boldon & "Name: " & @boldoff & txt & @newline

~output any race
txt = hero.firstchild["Race.?"].field[name].text
if (empty(txt) <> 0) then
  txt = "-none-"
  endif
append @boldon & "Race: " & @boldoff & txt & @newline

~output age
append @boldon & "Age: " & @boldoff & hero.child[mscPerson].field[perAge].text & @newline
```

Category: Kit Reference

Trigger Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"trigger" type, "pick" context

The Trigger script utilizes the following special symbols:

-None-	There are no special symbols for a Trigger script.

Description

The Trigger script is used in conjunction with the "trigger" action portal, which makes it possible for the user to invoke a one-time operation. The primary use of the Trigger script is for either resetting values (e.g. trackers on the In-Play tab) or applying adjustments to usage pools (e.g. damage tracking). In the latter case, there will typically be an edit portal where the user can enter a value. When the Trigger script is invoked, the value is used when applying the adjustment to the usage pool, and then the field is reset.

When invoked, a Trigger script starts with the pick as its initial context. There is generally no reason to navigate outside the context of the pick, but it is technically allowed. The primary focus will be on the pick and its fields.

Example

We'll use the Trigger script associated with the button to reset all damage on the In-Play tab as an example. In this script, the usage pools for damage tracking are reset.

```
~if there is no history to undo, notify the user
if (hero.usagepool[DmgNet].count = 0) then
  notify "Undo history is empty"
  done
  endif

~empty out both usage pools
perform hero.usagepool[DmgNet].empty
perform hero.usagepool[DmgAdjust].empty
```

Category: Kit Reference

Integrate Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Definition File
Procedure Use:	"integrate" type, "pick" context

The Integrate script utilizes the following special symbols:

-None-	There are no special symbols for a Integrate script.

Description

The Integrate script serves a very specific purpose. When the user triggers the integration of pending actors into an existing combat within the Tactical Console, this script allows the author to piggyback additional special handling. In general, you should not need to perform any special behaviors, but the mechanism is provided just in case.

When integration is triggered by the user, the Integrate script is applied to each actor **after** it has been properly integrated into the combat. When invoked, the Integrate script starts with the "actor" pick of an actor as its initial context.

Example

In a game system like D&D 4th Edition, there are "encounter" powers, which can be used once per encounter. It might make sense for a game like this to automatically reset the state of each encounter power within the Integrate script. If so, then the corresponding script might look like below.

```
foreach pick in hero where "Power.Encounter"
  eachpick.field[pwrIsUsed].value = 0
  nexteach
```

Category: Kit Reference

NewCombat Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Definition File
Procedure Use:	"newcombat" type, "pick" context

The NewCombat script utilizes the following special symbols:

isfirst	(Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

Description

The NewCombat script is invoked whenever the user triggers the start of a new combat within the Tactical Console. Prior to the actual transition into combat mode, this script is applied to each actor. When invoked, the NewCombat script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored), and that is done before the NewTurn or Initiative scripts are invoked. This allows the author to reset state for each actor prior to the new combat.

Example

In various game systems, there is the notion of "waiting" or "holding an action". This state will persist for each actor after combat ends, so it needs to be reset at the start of combat.

```
herofield[acAbandon].value = 0
```

Category: Kit Reference

EndCombat Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Definition File
Procedure Use:	"endcombat" type, "pick" context

The EndCombat script utilizes the following special symbols:

isfirst	(Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

Description

The EndCombat script is the counterpart of the NewCombat script and is invoked whenever the user triggers the end of a combat within the Tactical Console. When combat is ended, this script is applied to each actor. Like its counterpart, the EndCombat script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored). This allows the author to reset state for each actor after combat completes.

Example

In various game systems, there is the notion of "waiting" or "holding an action". This state will persist for each actor after combat ends, so it can be reset at the end of combat instead of at the start.

```
herofield[acAbandon].value = 0
```

Category: Kit Reference

NewTurn Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Definition File
Procedure Use:	"newturn" type, "pick" context

The NewTurn script utilizes the following special symbols:

isfirst	(Number) Entry: Non-zero if this script is being invoked for the first actor in the portfolio. Exit: Ignored.

Description

The NewTurn script is invoked whenever the user triggers the start of a new combat turn within the Tactical Console. Prior to the start of the new turn, this script is applied to each actor. When invoked, the NewTurn script starts with the "actor" pick of an actor in the combat as its initial context. It is invoked for all actors in the combat (non-combatants are ignored), being performed after any NewCombat script and before any Initiative script. This allows the author to reset state for each actor prior to each new combat turn.

Example

If a game system has state for each actor that needs to be reset at the start of each new combat turn, that can be accomplished via the NewTurn script.

```
herofield[acState].value = 0
```

Category: Kit Reference

Initiative Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	No
Where Used:	Definition File
Procedure Use:	None

The Initiative script utilizes the following special symbols:

initiative	(Number) Entry: The previous initiative value for the actor or zero if not yet specified. Exit: Calculated initiative value to use for the actor.
tiebreaker	(Number) Entry: The previous tie-breaker value for the actor or zero if not yet specified. Exit: Calculated tie-breaker initiative value to use for the actor.

Description

The Initiative script is used to generate appropriate initiative values for each actor. It is invoked whenever a new initiative value is needed, as determined by the configuration settings in the definition file. It is always invoked after any NewCombat or NewTurn script is invoked.

There are two separate initiative values that must be generated. The first is the standard initiative value used for the game system. Once it is calculated, it should be assigned to the "initiative" special symbol. The second initiative value is the tie-breaker to be used if the two actors have the exact same primary initiative score. For example, in the d20 System, the dexterity bonus is used as a tie-breaker. This value should be assigned to the "tiebreaker" special symbol.

When invoked, the Initiative script starts with the "actor" pick of an actor in the combat as its initial context. From there, you can access whatever facets of the character are necessary to properly calculate the initiative scores.

Example

In the d20 System, the initiative score is a d20 roll plus the initiative bonus, while the dexterity bonus is used as the tie-breaker. This yields a script that looks like the following.

```
~Initiative is a d20 roll (0 to 19 + 1) plus initiative bonus
@initiative = random(20) + 1 + hero.child[trInit].field[trtFinal].value
~Tie-breaker is the dexterity bonus
@tiebreaker = hero.child[attrDex].field[trtFinal].value
```

Category: Kit Reference

Merge Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	Pick
Fields Finalized?	Yes
Where Used:	Components
Procedure Use:	None

The Merge script utilizes the following special symbols:

-None-	There are no special symbols for the Merge script.

Description

The Merge script is used to handle the merging of two separate picks into one via stacking behavior. It is invoked whenever the user performs a merge operation of stackable gear. The goal of the script is to allow authors to properly reconcile whatever fields are necessary for an accurate merging of the two picks.

The Merge script is defined for a component. As such, it is inherited into every thing that derived from that component. Each script is intended to properly merge the fields within that component and nothing else. This ensures that a thing derived from multiple components, each with their own Merge script, will perform appropriate merge behavior without incident.

When invoked, the Merge script starts out with a pick as its initial context. However, there are two picks involved in the merge operation. When a merge takes place, there is the pick that is assimilating the other (e.g. increasing its quantity) and the pick that is ultimately deleted once the merge completes. The initial context is the pick that is being assimilated into. The second pick is made available via an alternate pick context, which can be accessed via the "altpick" context. This makes it possible to assimilate one pick into the other, regardless of what fields or behaviors are involved.

Example

The sample Merge script below adds the pertinent quantities of the pick being merged into the pick that is being assimilated into.

```
field[trkMax].value += altpick.field[trkMax].value
field[trkUser].value += altpick.field[trkUser].value
```

Category: Kit Reference

Split Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	Pick
Fields Finalized?	Yes
Where Used:	Components
Procedure Use:	None

The Split script utilizes the following special symbols:

-None-	There are no special symbols for the Split script.

Description

The Split script handles the splitting of one piece of gear into two separate picks via stacking behavior. It is invoked whenever the user performs a split operation of stackable gear. The goal of the script is to allow authors to properly reconcile whatever fields are necessary for an accurate splitting of one pick into two distinct picks.

The Split script is defined for a component. As such, it is inherited into every thing that derived from that component. Each script is intended to properly split the fields within that component and nothing else. This ensures that a thing derived from multiple components, each with their own Split script, will perform appropriate split behavior without incident.

When invoked, the Split script starts out with a pick as its initial context. However, there are two picks involved in the split operation. When a split takes place, there is the initial pick that is being split (e.g. decreasing its quantity) and the new pick that is ultimately created through the split process. The initial context is the existing pick that is being split. The second pick is made available via an alternate pick context, which can be accessed via the "altpick" context. This makes it possible to split one pick into two, regardless of what fields or behaviors are involved.

When splitting picks, the engine will automatically update the "stackQty" field of both picks with the appropriate values. Therefore, if a pick with a quantity of 20 is split, with a new quantity of 7 specified, the engine will setup the pick being split with a "stackQty" field of 13 and the new pick with a "stackQty" of 7. You can then use these field values to accurately adjust everything else for each pick.

Example

The sample Split script below splits one pick into two and properly adjusts all user-tracking details to accurately reflect the impact of the change.

```
~save the quantity of ammo that still remains unused
var user as number
user = field[trkUser].value

~update the new "max" values for both picks based on the new stack quantity
field[trkMax].value = field[stackQty].value
altpick.field[trkMax].value -= altpick.field[stackQty].value

~the user value for the first pick is the quantity of ammo still unused,
~subject to the maximum size for the pick
field[trkUser].value = minimum(user,field[trkMax].value)

~subtract the quantity allocated to the first pick from what's now left
user -= field[trkUser].value

~if we have any unused ammo left to assign to the other pick, assign it
if (user <= 0) then
  altpick.field[trkUser].value = 0
else
  altpick.field[trkUser].value = user
  endif
```

Category: Kit Reference

Change Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Container
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Portals
Procedure Use:	"container" context

The Change script utilizes the following special symbols:

-None-	There are no special symbols for the Change script.

Description

The Change script is utilized within chooser and menu portals. The purpose of the script is to enact special handling whenever the contents of the portal are changed.

It is rare that you'll need to use this mechanism, because the impact of a selection will usually be handled via Eval scripts during the next evaluation cycle. The key exception to this is when the user is customizing a gizmo within the form for that gizmo. If the influence of the selection is required before the gizmo changes are finally saved, a Change script is needed to apply the necessary effects.

As an example, consider the World of Darkness data files. When a character spends XP during advancement, the XP cost for each advancement varies based on a numerous conditions. The Change script is utilized to properly calculate the XP cost for the selected advancement, which can then be displayed to the user before the advancement is officially added.

When invoked, the Change script starts out with a container as its initial context. This container is the one that the portal is associated with. For example, a chooser portal will reside within a layout, which is within a panel or form. The container associated with the panel or form is the initial context.

Example

The Change script below shows how the calculation of XP cost is performed for the World of Darkness game system. This example is edited down to show the core behaviors, as the real script handles additional factors.

```
~get the cost of each dot for the ability
var eachdot as number
eachdot = firstchild["Advance.Gizmo"].field[idotcost].value

~determine the tagexpr with which to identify the pick we're interested in
var tagexpr as string
tagexpr = "component.CanAdvance & Advance." & firstchild["Advance.Gizmo"].idstring

~if there is a half-price discount for the ability, incorporate that
if (hero.tagsearch["HalfPrice." & firstchild["Advance.Gizmo"].idstring] > 0) then
  eachdot = eachdot / 2
  eachdot = round(eachdot,0,1)
  endif

~if there is a double-cost penalty for the ability, incorporate that
if (hero.tagsearch["DoubleCost." & firstchild["Advance.Gizmo"].idstring] > 0) then
  eachdot *= 2
  endif

~get the previous dot level
current = hero.firstchild[tagexpr].field[iprevlevel].value

~get the current dot level
nextlevel = hero.firstchild[tagexpr].field[ilevel].value

~determine the XP cost to improve to the next dot level
var xp as number
var dotcount as number
var dotcost as number
dotcount = nextlevel
call dotcost
xp = dotcost
```

```
dotcount = current
call dotcost
xp -= dotcost
xp = xp * eachdot

~save out the calculated XP cost
child[advDetails].field[ixpcalc].value = xp
child[advDetails].field[inewdots].value = nextlevel
```

Category: Kit Reference

TransactSetup Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick or Thing
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Components
Procedure Use:	"xactsetup" type, "transact" context, "pick" context

The TransactSetup script utilizes the following special symbols:

isbuy	(Number) Entry: Indicates whether the transaction is buy (non-zero) or sell (zero). Exit: Ignored.
ispick	(Number) Entry: Indicates whether the context is a pick (non-zero) or a thing (zero). Exit: Ignored.
special	(Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

Description

The TransactSetup script is invoked at the beginning of a transaction, whether it be a buy or sell operation. When invoked, the initial context is either the thing being purchased or the pick being sold. The script can also access the separate "transaction" context to manipulate the transaction pick. In fact, the role of the TransactSetup script is to appropriately configure the transaction pick so that everything is properly presented and handled for the user.

The engine will automatically setup the transaction pick with a few pieces of information. It's up to you to setup the remaining details for how you want the transaction to be handled. The automatic behaviors of the engine differs for buying and selling operations, so both are outlined separately below.

Setup logic for a "buy" transaction:

1. The "xactName" field is set to the name of the thing being purchased
2. The "xactLimit" field is set to no limit
3. The "xactQty" field is set to the lot size for the thing being purchased
4. The TransactSetup script is invoked

Setup logic for a "sell" transaction:

1. The "xactName" field is set to the name of the thing being purchased
2. The "xactLimit" field is set to the total quantity possessed for the pick
3. The "xactQty" field is set to the total quantity possessed
4. The TransactSetup script is invoked

Within the TransactSetup script, the "xactEach" field must always be set to the proper unit cost for the item being purchased or sold. Other fields may be setup as you deem appropriate to the transaction requirements.

Please see the separate documentation for further details on using transactions.

Example

The TransactSetup script below shows the behavior of the Sample data files for handling gear transactions. The "each" cost is setup to the cost of the gear, the actual amount paid is set to zero so that default handling is used, and the gear is identified as a holder if appropriate.

```
~start by assuming our unit cost is the cost of one item
var cost as number
```

```
cost = field[grCost].value

~setup the unit cost for the item
hero.transact.field[xactEach].value = cost

~zero out the cash amount to be paid (implying use of the standard cost)
hero.transact.field[xactCash].value = 0

~if the item is gear, setup whether the item holds other gear
if (isgear <> 0) then
  hero.transact.field[xactHolder].value = gearcount
  endif
```

Category: Kit Reference

TransactBuy Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Thing
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Components
Procedure Use:	"xactbuy" type, "transact" context, "pick" context

The TransactBuy script utilizes the following special symbols:

reject	(String) Entry: The empty string to indicate success. Exit: If non-empty, indicates the transaction was rejected, with the contents being shown to the user as the explanation for the failure.
special	(Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

Description

The TransactBuy script is invoked at the completion of a "buy" transaction. The purpose of the script is to appropriately complete the purchase. It achieves this by retrieving the details of the transaction from the special "transaction" pick. Once the information is retrieved, whatever actions comprise the purchase can be performed, such as subtracting the cost of the purchase from the cash possessed by the character.

When invoked, the initial context is the thing being purchased. However, this script will primarily be interested in the contents of the "transaction" pick, which can be accessed via the "transaction" context. The transaction pick will contain all of the particulars specified by the user for the purchase.

The contents of the transaction pick will depend entirely on the "buy" template utilized. Whatever portals are presented within the buy template will dictate the values of fields within the transaction pick.

The engine will retrieve the final quantity purchased from the "xactQty" field of the transaction pick. The value of this field will dictate the actual quantity assigned to the new pick that is purchased.

Please see the separate documentation for further details on using transactions.

Example

The TransactBuy script below shows the behavior of the Sample data files for handling gear transactions. If the gear is free, the transaction is completed without any changes to the character's cash. Otherwise, the appropriate cash is deducted from the proper usage pool, with a rejection error being reported if there is insufficient cash.

```
~if we're buying for free, no cash should be touched, so get out
if (hero.transact.field[xactIsFree].value <> 0) then
  done
  endif

~get the cash amount specified by the user
var cash as number
cash = hero.transact.field[xactCash].value

~if no cash amount was given, get the standard total cost for the purchase
if (cash = 0) then
  cash = hero.transact.field[xactQty].value * hero.transact.field[xactEach].value
  endif

~if we don't have enough cash to make the purchase, reject the transaction
if (cash > herofield[acCashNet].value) then
  @reject = "You lack sufficient cash to purchase the item."
  done
  endif

~subtract the amount from the current pool of cash
perform hero.usagepool[TotalCash].adjust[-cash]
```

Category: Kit Reference

TransactSell Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Pick
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Components
Procedure Use:	"xactsell" type, "transact" context, "pick" context

The TransactSell script utilizes the following special symbols:

reject	(String) Entry: The empty string to indicate success. Exit: If non-empty, indicates the transaction was rejected, with the contents being shown to the user as the explanation for the failure.
special	(Number) Entry: Value specified with the definition of the portal to allow different behaviors based on usage. Exit: Ignored.

Description

The TransactSell script is invoked at the completion of a "sell" transaction. The purpose of the script is to appropriately complete the sale. It achieves this by retrieving the details of the transaction from the special "transaction" pick. Once the information is retrieved, whatever actions comprise the sale can be performed, such as adding the proceeds of the sale to the cash possessed by the character.

When invoked, the initial context is the pick that is being sold. However, this script will primarily be interested in the contents of the "transaction" pick, which can be accessed via the "transaction" context. The transaction pick will contain all of the particulars specified by the user for the sale.

The contents of the transaction pick will depend entirely on the "sell" template utilized. Whatever portals are presented within the sell template will dictate the values of fields within the transaction pick.

The engine will retrieve the final quantity sold from the "xactQty" field of the transaction pick. The value of this field will dictate the actual quantity deducted from the pick that is sold. If the entire quantity of the pick is sold, the pick is deleted.

Please see the separate documentation for further details on using transactions.

Example

The TransactSell script below shows the behavior of the Sample data files for handling gear transactions. The cash value specified by the user is retrieved and added back to the usage pool the manages the character's cash.

```
~get the cash amount specified by the user
var cash as number
cash = hero.transact.field[xactCash].value

~add the amount to the current pool of cash
perform hero.usagepool[TotalCash].adjust[cash]
```

Category: Kit Reference

CanAdvance Script

Context: HL Kit ... Kit Reference ... Script Types

Technical Details

Initial Context:	Hero
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Definition File
Procedure Use:	None

The CanAdvance script utilizes the following special symbols:

message	(String) Entry: The empty string to indicate the character can transition to advancement mode. Exit: If non-empty, indicates the character cannot transition to advancement mode, with the contents being shown to the user as the explanation of what must still be completed for the character. The text may contain encoding.

Description

The CanAdvance script is only utilized when the formalized advancement mechanism is enabled for the game system. This script allows the author to verify that the minimum configuration has been completed for a character before allowing the user to switch from creation mode to advancement mode.

The CanAdvance script is invoked when the user attempts to switch to advancement mode via the menu. When the user tries to make the change, the script is triggered, at which point the script verifies that the necessary creation steps have been completed for the character. If any are not satisfied, an error message is displayed to the user and the user is not allowed to transition until the creation process is completed.

It is the responsibility of the script to synthesize the message shown to the user that reports the creation steps that have not been satisfactorily completed. If a problem is detected in the script, a suitable error message should be appended to the "message" special symbol. If multiple errors occur, the message should contain a list of them, with each message on a new line. If there are no errors, then the special symbol should be the empty string, and that tells HL that the character is ready to transition to advancement mode.

When invoked, a CanAdvance script starts with the actor as its initial context. You are free to check any aspect of the character when verifying that the transition to advancement mode is allowed.

NOTE! Don't be too picky with the CanAdvance script. Remember that every gaming group has its own house rules that will modify the basic game rules. As such, it's important to use validation rules to trap most errors instead of requiring the user to obey all rules. The CanAdvance script should only be used to enforce details that should always be required and/or that impact your ability to write quality data files.

Example

The CanAdvance script below shows the behavior of the Sample data files for handling the advancement test. The script sets up a bullet character to put at the start of each error and then performs the tests. If any test fails, a suitable error message is appended to the message.

```
var bullet as string
bullet = "{bmp bullet_red}{horz 4}"

~perform tests to assure starting resources have been assigned
if (#resleft[resCP] <> 0) then
  @message = @message & bullet & "Character points must be assigned for the character.\n"
  endif
if (#resleft[resAbility] <> 0) then
  @message = @message & bullet & "Ability slots must be assigned for the character.\n"
  endif
```

Category: Kit Reference

Transition Script

Context: HL Kit … Kit Reference … Script Types

Technical Details

Initial Context:	Hero
Alternate Context:	None
Fields Finalized?	Yes
Where Used:	Definition File
Procedure Use:	None

The Transition script utilizes the following special symbols:

message	(String) Entry: The empty string to indicate that no transition message should be shown to the user. Exit: If non-empty, specifies the transition message to be shown to the user as a reminder. The text may contain encoding.

Description

The Transition script is only utilized when the formalized advancement mechanism is enabled for the game system. This script allows the author to display a message to the user whenever the user successfully switches from creation mode to advancement mode, or vice versa. In general, the message is simply a reminder about the implications of the transition, as well as how to switch back. If you want to suppress the message, simply use an empty message string (or omit the script entirely). The message starts out centered, although you can change the behavior.

When invoked, a Transition script starts with the actor as its initial context. You are free to refer to any aspect of the character, although doing so is rarely necessary. The primary detail you'll want to check is whether the character is transitioning to creation mode or advancement mode, which can be determined via the "state" context.

Example

The Transition script below shows the standard behavior you'll want to include in your data files. Depending on the new mode, an appropriate message is displayed.

```
if (state.iscreate <> 0) then
  @message = "{b}{text ffff00}Creation Phase{text 010101}{/b}"
  @message &= "\n\n"
  @message &= "{align left}You have unlocked your character!"
else
  @message = "{b}{text ffff00}Advancement Phase{text 010101}{/b}"
  @message &= "\n\n"
  @message &= "{align left}You have locked your character creation traits. "
  endif
```

Category: Kit Reference

Definition File Reference

Context: HL Kit … Kit Reference

Overview

Each game system has a single definition file that describes the fundamental, non-changing characteristics of the game. HL uses the definition file to configure itself for each game system, reading in the definition file before any other files. All of the contents of the other data files are given meaningful interpretation by the contents of the definition file.

The definition file must have the file extension ".def". By convention, the file is always named "definition.def" so that it can always be readily identified.

This section outlines the structure and mechanics for writing a definition file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

Structural Composition

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Definition".

The following table defines the attributes for a "document" element.

signature	Text – Must be the value "Hero Lab Definition".

Within the document element, every definition file possesses the following child elements, appearing in the sequence given and with the names specified.

game	A single "game" element must appear as defined by the given link. This element contains basic game system information.
author	An optional "author" element may appear as defined by the given link. This element contains information about the author of the data files.
release	A single "release" element must appear as defined by the given link. This element provides details about the current release of the data files.
structure	A single "structure" element must appear as defined by the given link. This element specifies structural details about the game system and its behavior.
behavior	A single "behavior" element must appear as defined by the given link. This element details behavioral aspects for the game system and data files.
scriptmacro	Zero or more "scriptmacro" elements may appear as defined by the given link. This element defines macros for use within scripts.
advancement	An optional "advancement" element may appear as defined by the given link. This element contains details regarding the advancement works within the data files.
phase	Zero or more "phase" elements may appear as defined by the given link. This element dictates the set of evaluation phases that will exist for the game system.

Category: Kit Reference

Game Element (Data)

Context: HL Kit ... Kit Reference ... Definition File Reference

The "game" Element

The game system information section defines the name of the game system, the manufacturer, and copyright information. The XML element name is "game" and the complete list of attributes is below.

game	Text – Name of the game system. Maximum length is 50 characters.
publisher	Text – Name of the manufacturer or publisher of the game system. Maximum length is 50 characters.
website	Text – URL of the publisher’s web-site. Maximum length is 100 characters.
copyright	(Optional) Text – Appropriate copyright and trademark declaration for the game system and data files. Maximum length is 250 characters. Default: Empty.
legaloutput	Text – This is shortened legal text that is inserted at the bottom of all character sheet output. You must provide appropriate copyright and trademark declarations.
manualroot	(Optional) Text – Specifies the name of the file to be used as the User Manual for the game system, which is accessible via the Configure Hero form and the Help menu within HL. This should generally be an HTML file. Default: "docs\manual.htm".
editorroot	(Optional) Text – Specifies the name of the file used as the manual for working with the Editor when you data files are loaded. Within the Editor, this manual is accessible via the Help menu. This should generally be an HTML file. Default: "docs\editor.htm".

Example

The following example demonstrates what a "game" element might look like. All default values are assumed for omitted optional attributes.

```
<game
  name="Sample Game Systems"
  publisher="Game Company"
  website="www.gamecompany.com"
  copyright="Copyright 2008 by Game Company. Game XYZ is a trademark of Game Company."
  legaloutput="Copyright 2008 by Game Company.">
  </game>
```

Category: Kit Reference

Author Element (Data)

Context: HL Kit ... Kit Reference ... Definition File Reference

The "author" Element

The author information section contains details about the author of the data files and how best to contain the author regarding the data files. The XML element name used is "author" and the complete list of attributes is below.

author	Text – Name of the author. Maximum length is 50 characters.
email	(Optional) Text – Email address of the author to send questions and/or bug reports to. Maximum length is 70 characters. Default: Empty.
website	(Optional) Text – URL of the author’s web-site where a FAQ and/or useful information can be found. Maximum length is 100 characters. Default: Empty.

Example

The following example demonstrates what an "author" element might look like. All default values are assumed for optional attributes.

```
<author
  author="John Q. Author"
  website="www.johnqauthor.com">
  </author>
```

Category: Kit Reference

Release Element (Data)

Context: HL Kit … Kit Reference … Definition File Reference

The "release" Element

All details pertaining to the current release of the data files can be found here. The XML element name used is "release" and the complete list of attributes is below.

major	Integer – Major version number assigned to the release. Must be between 0-255.
minor	Integer – Minor version number assigned to the release. Must be between 0-255.
required	(Optional) Text – Specifies the minimum version of HL required to utilize these data files. The version is given in the exact same format displayed by the product when the "About Hero Lab" option is selected under the "Help" menu (e.g. “2.3a"). If empty, no requirements are enforced and the data files are assumed to work with all versions of the product. Default: Empty.
summary	(Optional) Text – Arbitrary text used as release summary notes to display to the user. This summary is used by the HLExport tool when packaging up your data files for distribution to other users. No maximum length. Default: Empty.
PCDATA	(Optional) Text – Detailed release notes for the data files can be specified within the PCDATA block of the element. These release notes will be displayed to the user every time the data files are re-compiled and then loaded. This is an ideal place to inform users about the status of the data files, new capabilities that have been added, and where to get answers to questions about using the data files.

Example

The following example demonstrates what a "release" element might look like. All default values are assumed for omitted optional attributes.

```
<release
  major="2"
  minor="1"
  summary="Release summary here"><![CDATA[These are the actual release notes.
  ]]></release>
```

Category: Kit Reference

Structure Element (Data)

Context: HL Kit ... Kit Reference ... Definition File Reference

The "structure" Element

Every game system will have an assortment of structural details that must be specified, and these are all grouped together within a "structure" element. The complete list of attributes for a structure element is below.

folder	Text – Unique name to be used for the folder in which all data files for this game system are placed. The name may consist only of alphanumeric characters (i.e. letters and digits), with no spaces or punctuation other than "_" and "-" allowed. Maximum length is 25 characters. WARNING! The folder name has critical implications and must be chosen carefully.
editwidth	(Optional) Integer – Specifies the fixed width to use for all edit (i.e. tab-based) panels within HL. The value is given in units of pixels. Default: "500".
summarymin	(Optional) Integer – Specifies the minimum width that will be allowed for all summary panels within HL. The value is given in units of pixels. Default: "135".
summarymax	(Optional) Integer – Specifies the maximum width that will be allowed for all summary panels within HL. The value is given in units of pixels. Default: "165".
heroterm	(Optional) Text – Name to be used when referring to a hero for this game system. Maximum length is 25 characters. Default: "hero".
thingterm	(Optional) Text – Name to be used when referring to a thing for this game system. Maximum length is 25 characters. Default: "thing".
entityterm	(Optional) Text – Name to be used when referring to an entity for this game system. Maximum length is 25 characters. Default: "entity".
combatturnterm	(Optional) Text – Name to be used when referring to a combat turn for this game system. Maximum length is 25 characters. Default: "turn".

The "structure" element also possesses child elements that describe additional facets of the game system. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

datetime	Zero or more "datetime" elements may appear as defined by the given link. This element defines rules for date and time composition.

The "datetime" Element

The "datetime" element dictates the structure of dates and times for the game system. The sequence in which the "datetime" elements are defined dictates the composition used within HL. By default, the date and time structures for gamespace are assumed to be those used in realspace. The complete list of attributes for a "datetime" element is below.

name	Text – Name to be used for the component of the date or time. Maximum length is 20 characters.
digits	Integer – Specifies the number of digits used to specify the date/time component.
istime	Boolean – Indicates whether this is a component of the game system's time or date.

Example

The following example demonstrates what a "structure" element might look like. All default values are assumed for optional attributes. Within this example, the composition of a game date is defined as if it were a date in realspace.

```
<structure
  folder="skeleton"
  editwidth="520"
  combatturnterm="round">
  <datetime name="month" digits="2"/>
  <datetime name="day" digits="2"/>
  <datetime name="year" digits="4"/>
  </structure>
```

Category: Kit Reference

Behavior Element (Data)

Context: HL Kit … Kit Reference … Definition File Reference

Contents

The "behavior" Element

Each game system has a variety of behaviors that will need to be defined. These behaviors are collectively specified within a "behavior" element. The complete list of attributes for a "behavior" element is below.

defaultname	(Optional) Text – Default name to be used for all actor that have not been explicitly named by the user. Maximum length is 50 characters. Default: "Unnamed".
secondaryphase	(Optional) Id – Specifies the unique id of the phase to be used as the default for all Secondary tag expressions assigned by tables and choosers. If no explicit phase is assigned for a Secondary tag expression, this one is used. Default: "Setup".
secondarypriority	(Optional) Integer – Specifies the priority to be used as the default for all Secondary tag expressions assigned by tables and choosers. If no explicit priority is assigned for a Secondary tag expression, this one is used. Default: "5000".
existencephase	(Optional) Id – Specifies the unique id of the phase to be used as the default for all Existence tag expressions assigned by tables and choosers. If no explicit phase is assigned for an Existence tag expression, this one is used. Default: "Setup".
existencepriority	(Optional) Integer – Specifies the priority to be used as the default for all Existence tag expressions assigned by tables and choosers. If no explicit priority is assigned for an Existence tag expression, this one is used. Default: "5000".
gearphase	(Optional) Id – Specifies the unique id of the phase during which all built-in gear processing is performed. This includes the calculation of gear weights throughout all gear holders. Default: "Effects".
gearpriority	(Optional) Integer – Specifies the priority at which all built-in gear processing is performed. Default: "5000".
mouseinfo	(Optional) Id – Specifies the unique id of the procedure to automatically invoke as the MouseInfo script within portals. Default: "MouseInfo".
description	(Optional) Id – Specifies the unique id of the procedure to automatically invoke as the Description script within portals. Default: "Descript".
initascend	(Optional) Boolean – Indicates whether initiative order ascends or descends. When it ascends, lower numbers take their actions first, and vice versa. Default: "no".
initperturn	(Optional) Boolean – Indicates whether to re-generate new initiative values at the start of every turn instead of at the start of every combat. Default: "no".
initsimultaneous	(Optional) Boolean – Indicates whether it is valid for two actors to have identical initiatives and act simultaneously. If not, HL will automatically generate a tie-breaker value. Default: "no".
initminimum	(Optional) Boolean – Specifies the minimum value allowed for the initiative value when the user is adjusting the value on the Tactical Console. Default: "-99".
initmaximum	(Optional) Boolean – Specifies the maximum value allowed for the initiative value when the user is adjusting the value on the Tactical Console. Default: "99".
interleave	(Optional) Boolean – Indicates whether the evaluation cycles of all actors within the context of the same lead are interleaved, which governs the behavior of minions and masters. If enabled, masters can influence changes upon their minions and minions can also influence changes upon their masters. If not enabled,

masters all fully evaluated before minions, so the influence can only occur in the downward direction. Default: "no".

The "behavior" element also possesses child elements that describe additional facets of the game system. The list of these child elements is below and must appear in the order shown. Click on the links to access the details for each element.

diceroller	A single "diceroller" element must appear as defined by the given link. This element customizes the integrated Dice Roller.
leadsummary	An optional "leadsummary" element may appear as defined by the given link. This element defines the LeadSummary Script. If omitted, default handling is employed.
newcombat	An optional "newcombat" element may appear as defined by the given link. This element defines the NewCombat Script. If omitted, default handling is employed.
newturn	An optional "newturn" element may appear as defined by the given link. This element defines the NewTurn Script. If omitted, default handling is employed.
integrate	An optional "integrate" element may appear as defined by the given link. This element defines the Integrate Script. If omitted, default handling is employed.
initiative	An optional "initiative" element may appear as defined by the given link. This element defines the Initiative Script. If omitted, default handling is employed.
initfinalize	An optional "initfinalize" element may appear as defined by the given link. This element defines the InitFinalize Script. If omitted, default handling is employed.
loaderror	An optional "loaderror" element may appear as defined by the given link. This element defines the LoadError Script. If omitted, default handling is employed.

The "diceroller" Element

The "diceroller" element defines characteristics of the integrated Dice Roller mechanisms when used with the game system. The complete list of attributes for a "diceroller" element is below.

mode	(Optional) Set – Specifies the initial dice rolling mode to start out in for the game system. Must be one of the following: ▪ totals – The dice rolled are added and that total is displayed. ▪ successes – Each die rolled is compared against the threshold for declaring a success and the number of successes is displayed. ▪ Default: "totals".
dietype	Integer – Specifies the default die type to be rolled.
quantity	(Optional) Integer – Specifies the default number of dice to be rolled. Default: "1".
success	(Optional) Integer – Specifies the threshold value at which a success result is achieved. Game systems like World of Darkness and Shadowrun utilizes successes instead of adding the dice results. Default: "1".
explode	(Optional) Integer – Specifies the threshold value at which a success result "explodes" by allowing a re-roll. If a value of zero is given, no explosion occurs. Default: "0".
maxexplode	(Optional) Integer – Specifies the maximum number of times that a single roll may explode. A value of zero indicates that explosion is unlimited. Default: "0".

The "leadsummary" Element

The "leadsummary" element defines the LeadSummary script that is used to synthesize summary information for each actor. This summary is displayed when the user previews the available actors for import into the current portfolio. The complete list of attributes for a "scriptmacro" element is below.

PCDATA	Script – Specifies the code comprising the LeadSummary script.

The "newcombat" Element

The "newcombat" element defines the NewCombat script that is invoked whenever the user begins a new combat within the Tactical

Console. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the NewCombat script.

The "newturn" Element

The "newturn" element defines the NewTurn script that is invoked whenever the user starts a new turn of combat within the Tactical Console. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the NewTurn script.

The "integrate" Element

The "integrate" element defines the Integrate script that is invoked whenever the user integrates pending actors into an existing combat within the Tactical Console. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the Integrate script.

The "initiative" Element

The "initiative" element defines the Initiative script that is used to calculate the initiative score for each actor during combat. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the Initiative script.

The "initfinalize" Element

The "initfinalize" element defines the InitFinalize script that is used to tailor the final displayed text for the initiative score of each actor during combat. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the InitFinalize script.

The "loaderror" Element

The "loaderror" element defines the LoadError script that is used to provide helpful information to the user about changes across releases of your data files and errors that can be safely ignored. The complete list of attributes for the element is below.

PCDATA Script – Specifies the code comprising the LoadError script.

Example

The following example demonstrates what a "behavior" element might look like. All default values are assumed for omitted optional attributes.

```
<behavior
  defaultname="Nobody"
  interleave="yes">
  <diceroller mode="totals" dietype="6" quantity="1"/>

  <leadsummary><![CDATA[
    ~start with the race
    var txt as string
    txt = hero.firstchild["Race.?"].field[name].text
    if (empty(txt) <> 0) then
      txt = "No Race"
      endif
    @text &= txt

    ~append the CP rating of the character
    @text &= " - CP: " & hero.child[resCP].field[resMax].value
    ]]></leadsummary>

  <initiative><![CDATA[
    ~calculate the secondary initiative rating
    @secondary = 0
    ~calculate the primary initiative rating
    @initiative = random(10) + 1
    ]]></initiative>

  </behavior>
```

Category: Kit Reference

ScriptMacro Element (Data)

Context: HL Kit ... Kit Reference ... Definition File Reference

The "scriptmacro" Element

The "scriptmacro" element defines a macro for use within the scripting language to conveniently access object details. Each macro specifies a kind of shorthand notation for a lengthy block of script code that can be used in place of the full code and will be treated as if the full code were entered. Script macros are globally defined and may be used within any script. The complete list of attributes for a "scriptmacro" element is below.

name	Text – Name to be used for the macro. May consist solely of alphanumeric characters.
result	Text – The resulting text to be generated when the macro is used. Each parameter is spliced into the result appropriately.
param1	(Optional) Text – Name used for the first parameter provided to macro. Used for splicing the value into the "result" given above.
param2	(Optional) Text – Name used for the second parameter provided to macro. Used for splicing the value into the "result" given above.
param3	(Optional) Text – Name used for the third parameter provided to macro. Used for splicing the value into the "result" given above.
param4	(Optional) Text – Name used for the fourth parameter provided to macro. Used for splicing the value into the "result" given above.
param5	(Optional) Text – Name used for the fifth parameter provided to macro. Used for splicing the value into the "result" given above.

Example

The following example demonstrates what a "scriptmacro" element might look like. All default values are assumed for omitted optional attributes.

```
<scriptmacro
  name="trait"
  param1="trait"
  result="hero.child[#trait].field[trtFinal].value">
  </scriptmacro>
```

Category: Kit Reference

Advancement Element (Data)

Context: HL Kit … Kit Reference … Definition File Reference

Contents

The "advancement" Element

Some game systems require that each new advancement of a character be performed in a rigid sequence. This is because the cost of increasing individual facets of the character go up as the character gets better. When a game system requires structured advancement, the "advancement" element can be defined to specify fundamental behaviors associated with the advancement. The complete list of attributes for an advancement element is below.

enable	(Optional) Boolean – Indicates whether a serialized advancement mechanism is used for the game system. Default: "no".
creationterm	(Optional) Text – Specifies the term presented to the user when referring to the initial mode where a new character is being created. Default: "creation".
advancementterm	(Optional) Text – Specifies the term presented to the user when referring to the post-creation mode where a character's advancement is serialized. Default: "advancement".

The "advancement" element also possesses child elements that describe additional facets of the advancement behavior. The list of these child elements is below and must appear in the order shown. Click on the links to access the details for each element.

canadvance	An optional "canadvance" element may appear as defined by the given link. This element defines the CanAdvance Script. If omitted, default handling is performed.
transition	An optional "transition" element may appear as defined by the given link. This element defines the Transition Script. If omitted, default handling is performed.

The "canadvance" Element

The "canadvance" element defines the CanAdvance script that determines whether a character satisfies the initial creation requirements for transitioning into serialized advancement. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the CanAdvance script.

The "transition" Element

The "transition" element defines the Transition script that is invoked when a character transitions between creation and advancement modes. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Transition script.

Example

The following example demonstrates what an "advancement" element might look like. All default values are assumed for omitted optional attributes.

```
<advancement
  enable="yes"
  creationterm="starting"
  advancementterm="improving">
  <canadvance><![CDATA[
    if (#resleft[resCP] <> 0) then
      @message = "Character does not meet requirement."
      endif
    ]]></canadvance>

  <transition><![CDATA[
    if (#iscreate[@newmode] = 1) then
      @message = "Now Creating!"
    else
      @message = "Now Advancing!"
      endif
    ]]></transition>

  </advancement>
```

Category: Kit Reference

Phase Element (Data)

Context: HL Kit … Kit Reference … Definition File Reference

The "phase" Element

Each phase within the evaluation cycle must be individually specified and the sequence in which the phase elements are defined dictates the sequence in which phases are processed during evaluation. The XML element name is "phase" and the complete list of attributes is below.

id	Id – Specifies the unique id assigned to this phase. This id is used to reference the phase throughout the data files.
name	Text – Common name for the phase is displayed in various situations. Maximum length is 25 characters.
description	(Optional) Text – Textual description of the role the phase serves within the game system. Default: Empty.
interleave	(Optional) Boolean – Indicates whether all tasks within this phase are either (a) interleaved across masters and minions or (b) processed hierarchically, with masters always being evaluated before minions. This works the exact same way as the "interleave" attribute within the "behavior" element, except that it only applies to tasks within this specific phase. Default: "yes". IMPORTANT! This attribute is only applicable when interleaving is **enabled** for the game system. It provides a means of forcing individual phases to be non-interleaved, which makes it possible to safely control dependencies of minions upon their masters for enablement conditions and the like. The default is "yes" so that all phases are interleaved by default if the game system behavior is designated as interleaved.

Example

The following example demonstrates what a "phase" element might look like. All default values are assumed for omitted optional attributes.

```
<phase
  id="Initialize"
  name="Initialization"
  description="Anything that has to happen before everything else">
  </phase>
```

Category: Kit Reference

Structural File Reference

Context: HL Kit ... Kit Reference

Overview

Structural files are where you'll be defining all of the pieces that provide the underlying framework for the game system. They are loaded immediately after the definition file.

The Kit supports four different structural file types that share the identical contents. They possess the file extensions ".1st", "core", ".str", and ".aug". These files are always loaded in that same order.

This section outlines the structure and mechanics for writing a structural file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

IMPORTANT! Just because you **can** put numerous different elements in the same file does not mean you **should** do so. Keeping your data files small and focused will also keep them much more manageable, so break up all the information across files where appropriate. See the Skeleton data files for examples of this.

Structural Composition

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Structure".

The following table defines the attributes for a "document" element.

signature	Text – Must be the value "Hero Lab Structure".

Within the document element, every structural file possesses the following child elements, appearing in the sequence given and with the names specified.

enmasse	Zero or more "enmasse" elements may appear as defined by the given link. This element specifies categories of things to automatically add to every actor.
bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies specific things to automatically add to every actor.
autoadd	Zero or more "autoadd" elements may appear as defined by the given link. This element identifies choices that are pre-selected for every actor.
usagepool	Zero or more "usagepool" elements may appear as defined by the given link. This element specifies usage pools for use by the game system.
reference	Zero or more "reference" elements may appear as defined by the given link. This element details references used throughout the data files.
group	Zero or more "group" elements may appear as defined by the given link. This element defines a collection of tag groups and their tags.
source	Zero or more "source" elements may appear as defined by the given link. This element specifies sources that can be configured by the user for each actor.
resource	Zero or more "resource" elements may appear as defined by the given link. This element defines resources like fonts and bitmaps used by styles for visual display.
style	Zero or more "style" elements may appear as defined by the given link. This element specifies an assortment of visual styles that can be used by portals.
component	Zero or more "component" elements may appear as defined by the given link. This element specifies a variety of components that can be used by component sets.
compset	Zero or more "compset" elements may appear as defined by the given link. This element defines various component sets that can be used to define things.
entity	Zero or more "entity" elements may appear as defined by the given link. This element defines entities that can be used within the game system.
sortset	Zero or more "sortset" elements may appear as defined by the given link. This element defines various sort sets that can be used by the game system.

Category: Kit Reference

EnMasse Element (Data)

Context: HL Kit ... Kit Reference ... Structural File Reference

The "enmasse" Element

For every game system, there will be related things that must be added to every actor. These things can be identified through the use of the "enmasse" element. The complete list of attributes for this element is below.

PCDATA	Text – This tag expression identifies the group of related things that should be added to every actor. Every thing that satisfies the tag expression is added.

The "enmasse" element also possesses child elements that pertain to the automatically added things. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

autotag	A single "autotag" element may appear as defined by the given link. This element specifies a tag that is automatically assigned to each added thing.

Example

The following example demonstrates what an "enmasse" element might look like. All default values are assumed for optional attributes.

```
<enmasse><![CDATA[component.Attribute]]>
  <autotag group="TheGroup" tag="TheTag"/>
  </enmasse>
```

Category: Kit Reference

AutoTag Element (Data)

The "autotag" Element

In a variety of situations, you will be able to specify one or more tags that will be automatically assigned to an object when it is added. This will always be controlled by the mechanism that is adding the object, such as bootstrapping a thing or adding a thing via a particular table.

Each tag that is to be automatically assigned is identified through the use of the "autotag" element. The complete list of attributes for this element is below.

group	Id – Specifies the unique id of the tag group within which the assigned tag is defined.
tag	Id – Specifies the unique id of the assigned tag within the above tag group.

Example

The following example demonstrates what a "autotag" element might look like. All default values are assumed for optional attributes.

```
<autotag group="TheGroup" tag="TheTag"/>
```

Category: Kit Reference

Bootstrap Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

Contents

The "bootstrap" Element

In a variety of situations, you will be able to specify individual things that should be automatically added to a container as picks. This process is called bootstrapping and has a diverse set of special rules that must be observed. Each bootstrapped thing is specified through the use of a "bootstrap" element. The complete list of attributes for this element is below.

thing	Id – Specifies the unique id of the thing that is to be automatically added to the container.
index	(Optional) Integer – Specifies a unique index value for this bootstrap entry. Only global bootstraps that add things to all actors require this attribute and it can be omitted in all other situations. Default: "0". WARNING! The index is used to uniquely identify this specific bootstrap within saved portfolios. Consequently, it must **never** be changed or re-used within different versions of the data files. If you eliminate a bootstrap entry, simply leave that index value unused, replacing the old "bootstrap" element with a comment explaining why the hole in numbering is left.

The "bootstrap" element also possesses child elements that pertain to the automatically added things. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the bootstrap is assigned to all. IMPORTANT! This element is only applicable when the bootstrap is defined directly within a component. In all other cases, this element may not be specified.
containerreq	An optional "containerreq" element may appear as defined by the given link. This element defines the conditional requirements that determine whether the bootstrap is made available within the container. The tag expression is applied against the container of the prospective bootstrapped pick. If the container does not satisfy the tag expression, the bootstrapped pick is treated as non-live. IMPORTANT! This element is not applicable when the bootstrap is defined within an entity.
autotag	Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to the added thing. IMPORTANT! When the bootstrap include a ContainerReq test, the tags are only assigned to the pick when ContainerReq tag expression is satisfied.
assignval	Zero or more "assignval" elements may appear as defined by the given link. This element specifies modified field values that are automatically assigned to the added thing.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing receives the specified bootstrap. The tag expression is applied against each thing derived from the component, and the bootstrap is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Match tag expression.

The "assignval" Element

All of the initial field values for a pick are dictated by the thing. However, there are situation where you'll want to dictate custom field values when bootstrapping a thing. For example, if you have a "Fly" ability, there may be different flying speeds. The "Fly" ability will have its initial speed, but each situation where you bootstrap the ability may call for a different speed.

To handle this situation, the "assignval" element allows you to designate the initial value to be used for a field within a bootstrapped pick. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field for which the value is being assigned.
value	Text – Specifies the new starting value to be assigned to the field. If the field is text-based, the value is simply assigned, although it may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.
behavior	(Optional) Set – Designates the assignment behavior to be used if the thing is unique and has already been added to the container. An ability like "Fly" is likely to be unique, so it's possible that the ability is bootstrapped by multiple sources or even user-selected. When this occurs, you can stipulate the rules for how the new value is assigned to the field. Must be one of these values: ▪ assign – The new value overrides any existing value. ▪ minimum – Use the lowest of the new value and any existing value. ▪ maximum – Use the highest of the new value and any existing value. ▪ Default: "assign".

IMPORTANT! There are a variety of important limitations that apply to the assignment of field values via bootstraps. These include the following:

- Non-persistent, derived fields may have their values specified freely.
- All other derived fields and all non-user fields may not be assigned via bootstraps.
- No array-based or matrix-based fields may be assigned via bootstraps.
- User fields may be assigned, but there are additional restrictions (see below).
- Value assignment only occurs for bootstraps that satisfy all their ContainerReq tests.
- If multiple bootstraps attempt to assign the same field value to a unique pick, the values are assigned in the order they are processed by the engine, which is dictated by evaluation timing.
    - If two or more value assignments are applied at the exact same timing, then there is *no* guarantee which will take precedence over the other.
    - If a bootstrap has ContainerReq tests, the value assignment is applied when the final condition test is resolved for the pick.

Fields of type "user" may be assigned via bootstraps. However, user field incur a number of additional restrictions, as outlined below:

- Cannot be used with conditional bootstraps (only derived fields may)
- Cannot be used with unique things, since the assigned value could overwrite an existing user-modified value
- Specified value is initialized as the default user value when the pick is first added and is thereafter allowed to be modified by the user
- The "maximum" and "minimum" behaviors are pointless with user values, since the value will always be the initial value assigned

Example

The following example demonstrates what a "bootstrap" element might look like. All default values are assumed for optional attributes.

```
<bootstrap thing="AttrIncr">
  <containerreq phase="Setup" priority="500">
    val:Level.? >= 4
    </container>
  <autotag group="TheGroup" tag="TheTag"/>
  <assignval field="MyField" value="42"/>
  </bootstrap>
```

Category: Kit Reference

ContainerReq Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

The "containerreq" Element

There will be times when you need to establish a condition for a thing where its container must satisfy a set of criteria. The criteria are spelled out via a Container tag expression. The tag expression is applied against the prospective container of the thing. If the container does not satisfy the tag expression, then the thing must be treated as if it simply doesn't exist within the container. Each separate set of requirements is specified through the use of a "containerreq" element. The complete list of attributes for this element is below.

phase	Id – Specifies the unique id of the evaluation phase during which the tag expression is tested.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested.
name	(Optional) Text – Specifies the name assigned to this test for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this test. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Container tag expression.

The "containerreq" element also possesses child elements that pertain to the requirements upon the container. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the container requirement test is applied to them all. IMPORTANT! This element is only applicable when the container requirement is defined within a component. In all other cases, this element may not be specified.
before	Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.
after	Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing is subject to the container requirement. The tag expression is applied against each thing derived from the component, and the requirement is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Match tag expression.

Example

The following example demonstrates what a "containerreq" element might look like. All default values are assumed for optional attributes.

```
<containerreq phase="Setup" priority="500" name="MyTest>
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
  val:Level.? >= 4
  </containerreq>
```

Category: Kit Reference

AutoAdd Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "autoadd" Element

There will likely be situations where you want to pre-select the contents of a chooser or pre-populate items into a table for the user. However, you also want the user to able to delete or replace these items. This is one of the various mechanisms for automatically adding picks to actors and it can be utilized via the "autoadd" element. The complete list of attributes for this element is below.

thing	Id – Specifies the unique id of the thing that is to be automatically added to the container.
portal	Id – Specifies the unique id of the portal into which the thing is to be added.

Example

The following example demonstrates what an "autoadd" element might look like. All default values are assumed for optional attributes.

```
<autoadd thing="journal" portal="journal"/>
```

Category: Kit Reference

UsagePool Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "usagepool" Element

Usage pools provide a convenient way to manage resources that have built-in history tracking of all changes. You define each usage pool via the "usagepool" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id to utilize for this usage pool.
name	Text – Name to be used in reference to this usage pool. Maximum length is 25 characters.
abbrev	Text – Abbreviation to be used for the usage pool. If empty, the name is also used as the abbreviation. Maximum length is 25 characters. Default: Empty.
quantity	Integer – Specifies the starting quantity to be used for the usage pool. Default: "0".
maximum	Integer – Specifies the maximum history size to be tracked for the pool. A value of zero indicates no limit on the history. Default: "0".
ishero	Boolean – Indicates whether the usage pool is intended for use on each hero/actor or with individual picks. Default: "yes".
persist	Boolean – Indicates whether the history information is saved within the portfolio and restored when it is reloaded. If not persistent, only the latest value is preserved. Default: "no".

Example

The following example demonstrates what a "usagepool" element might look like. All default values are assumed for optional attributes.

```
<usagepool
  id="TotalXP"
  name="Total XP"
  abbrev="XP"
  ishero="yes"
  persist="yes"/>
```

Category: Kit Reference

Reference Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "reference" Element

References provide a mechanism that is very similar to tool tips, with the mouse cursor changing to indicate that help is available for the region beneath the mouse. When the user clicks on the reference, the tip is displayed until the user clicks again to discard it. References are designed for use within encoded text and are ideal for annotating keywords and icons used within the data files. You can define a reference via the "reference" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id to utilize for this reference. **NOTE!** Unlike most other elements, the unique ids assigned to references may only possess a maximum of 9 characters instead of the usual 10.
contents	Text – Specifies the text to be inserted anywhere the reference is utilized. May contain encoded text.

NOTE! The use of references is not verified at compile time. It is only checked when text containing the reference is rendered in some way. If a reference is undefined when accessed, the text "[Undefined reference: name]" is output. In addition, you can view a list of any undefined references by going to the "Debug" menu and selecting "View Undefined String Usage".

Example

The following example demonstrates what a "reference" element might look like. All default values are assumed for optional attributes.

```
<reference
  id="BulletPrf"
  contents="Bullet-Proof Armor"/>
```

Category: Kit Reference

Group Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "group" Element

One of the cornerstones of describing objects in the Kit is the tags mechanism. All tags belong to named tag groups, with each tag group having a defined set of valid tag values that can be used. Tag groups also have a number of characteristics which dictate how the tags within that group are handled. Each tag group is specified through the use of a "group" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the tag group. This id is used in all references to the tag group.
dynamic	(Optional) Boolean – Indicates whether the tags for this group can be dynamically defined "on-the-fly" by specifying the tag directly on a thing. If a tag group is not dynamic, then all tags for the group must be defined within the group prior to their use within data files. If a tag group is dynamic, new tags can be defined throughout the data files by simply including the new tag id within a "tag" element on a thing. Dynamic tag groups cannot be assigned "explicit" sequencing. Default: "no".
sequence	(Optional) Set – Designates the sorting sequence to be used for the tags within the group. When you specify the tag group within a sort set, this sequence is used. Must be one of these values: ▪ ascii – Sort using ASCII codes (digits, all upper case, all lower case). ▪ nocase – Sort ignoring case (digits, upper case A, lower case A, upper B, etc.). ▪ textvalue – Treat the name of each tag as a numeric value (if no value, treated as zero for sorting). ▪ idvalue – Treat the id of each tag as a numeric value (if no value, treated as zero for sorting). ▪ explicit – Sort on the value of the "order" attribute assigned to each tag. ▪ id – Sort on the unique id of each tag as if it were text (allows handling of absolutely any situation). ▪ Default: "ascii".
minvalue	(Optional) Integer – Specifies the starting value to use for automatically generating value-based tags (see below). Default: "0".
maxvalue	(Optional) Integer – Specifies the starting value to use for automatically generating value-based tags (see below). Default: "0".

NOTE! Tag groups can automatically generate "value" tags for a given integer range via use of the "minvalue" and/or "maxvalue" attributes. Value tags possess a unique id that is simply an integer value, such as "42". Specifying either or both of these attributes as non-zero makes it easy to create a group of tags number X-Y (e.g. 1-42). When value tags are created, the group automatically uses "idvalue" sequencing. You can define explicit tags for the group that are blended with the auto-generated tags, and any author-defined tag takes precedence over whatever would be auto-generated. Lastly, if the group is dynamic, then additional tags can be defined on-the-fly, as needed.

NOTE! Dynamic tags are processed in the order they are encountered when the data files are compiled, and there is no guarantee of the order in which dynamic tags will be processed. If a dynamic tag has already been defined when it is next encountered, the original definition is used. This means that the first definition of a dynamic tag that happens to be encountered during compilation is the one that will be used for all instances of the tag, so you should strive to ensure that all dynamically defined tags have identical specifications.

The "group" element also possesses child elements that define the individual tags of the group. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

value	Zero or more "value" elements may appear as defined by the given link. This element specifies the tags that exist for the group. **NOTE!** While it is valid to specify zero tags here, every tag group must have at least one tag defined. You can only specify zero tags here if the group is dynamic and at least one tag is defined directly on a thing.

The "value" Element

The "value" element defines a tag for the containing tag group. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the tag within the group. This id is used in all references to the tag.

name	(Optional) Text – Name assigned to the tag. If empty, the unique id is used as the name. Maximum length is 100 characters. Default: Empty.
order	(Optional) Integer – Specifies the explicit order value to be used when sorting this tag against others within the tag group. This attribute only applies when the tag group is assigned the "explicit" sequence.

Example

The following example demonstrates what a "group" element might look like. All default values are assumed for optional attributes.

```
<group id="Equipment" dynamic="yes">
  <value id="Hand" name="Requires one or more hands"/>
  <value id="TwoHand" name="Requires two hands"/>
  <value id="AutoEquip" name="Gear is auto-equipped"/>
  <value id="Natural" name="Natural weapon/armor/etc."/>
  <value id="CustomGear" name="Custom Gear"/>
  </group>
```

Category: Kit Reference

Source Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "source" Element

Authors can use the sources mechanism as the mean for defining configuration settings that the user can toggle on and off via the "Configure Hero" form. When a given source is enabled by the user, a corresponding tag is automatically added to the actor, which can then be tested within the data files. Each individual source can be defined via the "source" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id to utilize for this source.
name	Text – Name to be displayed to the user for this source. Maximum length is 50 characters.
abbrev	(Optional) Text – Shortened name to be displayed within the summary of enabled sources. If empty, the name is used as the abbreviation. Maximum length is 50 characters. Default: Empty.
description	(Optional) Text – Description of the source and what behavior(s) it governs within the data files. Default: Empty.
parent	(Optional) Id – Specifies the unique id of a different source that is treated as the parent of this source. All sources are displayed in a hierarchy within the "Configure Hero" form. If empty, this source is presented to the user as a top-level selection. Default: Empty.
sortorder	(Optional) Integer – Assigns an explicit sort order to this source, enabling the display sequence of sources to be controlled by the author (see below). All sources are sequenced in increasing sort order. Default: "99999999".
selectable	(Optional) Boolean – Indicates whether the source is selectable by the user. This option allows sources that are solely used to visually group sources in a hierarchy to be made non-selectable. Default: "yes".
maxchoices	(Optional) Integer - Specifies the maximum number of child sources that can be selected by the user at once. This makes it possible to limit the user to select only one option from a list of sources. If zero, there is no limit to the number of sources that can be selected. Default: "0".
minchoices	(Optional) Integer - Specifies the minimum number of child sources that must be selected by the user beneath this source. This makes it possible to require the user to select one or more options from a list of sources. If zero, there is no requirement to select any sources. Default: "0".
reportable	(Optional) Boolean – Indicates whether the source is included in the hierarchical summary report of selected sources. Allows authors to prune the summary report of unnecessary layers so that the informnation presented to users is both clear and reasonably compact. Default: "yes".
reportname	(Optional) Text – Alternate name to be displayed for the source within the hierarchical summary report. Allows for a more compact name to be used for this purpose. If empty, the name of the source is used in the report. Default: Empty.
default	(Optional) Boolean – Indicates whether the initial (default) state of the source for a new actor is selected or non-selected. Default: "no".

NOTE! Sources that are user-selectable may **not** also possess child sources. If a user-selectable source is designated as the parent of another source, a compilation error will be reported.

NOTE! When either "minchoices" or "maxchoices" is specified as non-zero, child sources may NOT possess child sources of their own.

NOTE! By default, all sources are sorted alphabetically, based on their name. If an explicit sort order is specified, sources are sorted in increasing order. Sources with children are always sorted AFTER sources without children within the displayed list. This ensures that sources without children are grouped together and directly beneath the parent source for intuitive visual grouping. The handling of sources with/without children takes precedence over any explicit ordering that may be given.

Example

The following example demonstrates what a "source" element might look like. All default values are assumed for optional attributes.

```
<source
  id="Output"
  name="Output Options"
  selectable="no"
  description="Assortment of options governing character sheet output">
  </source>

<source
  id="Validation"
  name="Include Validation Report on Sheet"
  abbrev="Show Validation Report"
```

```
   parent="Output"
   default="yes"
   description="Whether validation report is included at bottom of character sheet">
   </source>
```

Category: Kit Reference

Resource Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

Contents

The "resource" Element

Everything associated with fonts, colors, bitmaps and borders is managed via resources. Resources can be defined as top-level elements within structural files. However, they can also be defined as child elements within styles. This latter technique will be more commonly used for bitmaps, since those bitmaps can be defined in conjunction with the style that uses the bitmap (e.g. portals used as buttons and icons).

Each visual attribute used within your data files is specified through the use of a "resource" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the resource. This id is used in all references to the resource.
isbuiltin	(Optional) Boolean – Indicates whether the resource is a "built-in" resource provided by HL for easy re-use. Only bitmaps and borders can be built-in resources, since fonts and colors can be freely defined at any time. Default: "no".
issystem	(Optional) Boolean – Indicates whether the resource is intended to replace a "system" resource utilized by HL. When you want to completely change the visual look of your data files and have that new look integrated into HL's own forms, you will need to specify system resources. Only specific resource ids can be replaced as system resources. Default: "no".

The "resource" element also possesses child elements that define the specifics of the resource. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! Exactly **one** of these child elements may be specified for each resource. If multiple are given, a compiler error will be reported.

color	An optional "color" element may appear as defined by the given link. This element specifies the details of a color resource.
font	An optional "font" element may appear as defined by the given link. This element specifies the details of a font resource.
bitmap	An optional "bitmap" element may appear as defined by the given link. This element specifies the details of a bitmap resource.
solid	An optional "solid" element may appear as defined by the given link. This element specifies the details of a solid-color border resource.
border	An optional "border" element may appear as defined by the given link. This element specifies the details of a bitmap-based border resource.

The "color" Element

The "color" element defines the facets of a color resource. The complete list of attributes for this element is below.

Text – Color value to be used in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a

color	Blue value of "80". **NOTE!** For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.

The "font" Element

The "font" element defines the facets of a font resource. The complete list of attributes for this element is below.

face	(Optional) Text – Name of the font face to use. This font must exist on every user's computer, or your data files will not successfully load. Consequently, it is generally a good idea to restrict yourself to the fonts that are provided with every copy of Windows. For example, you might limit yourself to using either "Arial" or "Times New Roman" for guaranteed support of all HL users. Default: "Arial".
size	(Optiona) Integer – Point size of the font to be used. The size is specified in "quarter points", allowing for fractional font sizes to be defined. Each increment of one equates to 0.25 points of font height. For example, to specify a font size of 10, you would use a value of 40 (10 times 4). For a font size of 10.5, use a value of 42. **NOTE!** Smaller point sizes become indistinguishable from each other when rendered on the screen. For example, there is no visible difference between a size of 16 and 17 (4.0 vs. 4.25 points) on the screen.
style	(Optional) Text – Specifies the font styles to be used for the resource. The various styles available are "bold", "italics", and "underline". You can combine multiple styles by placing a '+' between them (e.g. "bold+italics"). If empty, the "normal" version of the font is utilized, with no special styles applied. Default: Empty.
rotation	(Optional) Set – Designates the rotation angle at which the text will be drawn in this font. Note that rotated text is not always supported in all situations. In addition, all default sizing of portals assumes no rotation is employed, so you'll need to perform your own proper sizing when you use rotated text. Must be one of the following: ▪ 0 – No rotation is applied. ▪ 45 – Text is rotated 45 degrees. ▪ 90 – Text is rotated 90 degrees. ▪ 135 – Text is rotated 135 degrees. ▪ 180 – Text is rotated 180 degrees. ▪ 225 – Text is rotated 225 degrees. ▪ 270 – Text is rotated 270 degrees. ▪ 315 – Text is rotated 315 degrees. ▪ Default: "0"

The "bitmap" Element

The "bitmap" element defines the facets of a bitmap resource. The complete list of attributes for this element is below.

bitmap	Text – Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
istransparent	(Optional) Boolean – Indicates whether the bitmap should be treated as having built-in transparency. Enabling transparency allows you to have bitmaps that appear to be non-rectangular in shape (see below). Default: "no"

NOTE! Transparent bitmaps must be created appropriately so that they will behave transparently. Within the bitmap image, the pixel in the top left corner (position 0,0) is always considered to be transparent. All other pixels within the bitmap that possess the same color are also treated as transparent. The result is a bitmap that defines its own transparency.

The "solid" Element

The "solid" element defines the facets of a solid-color border resource. The complete list of attributes for this element is below.

color	Text – Color value to be used in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80".

	NOTE! For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.
thickness	(Optional) Integer – Specifies the thickness of the border in terms of pixels. Default: "1"

The "border" Element

The "border" element defines the facets of a bitmap-based border resource. Bitmap-based borders are comprised of eight bitmaps. There is one bitmap in each of the four corners around the visual element, plus four additional bitmaps that are used as a repeating pattern along each edge. Each bitmap may also be specified with a mask that is used to draw non-rectangular portions of the bitmap appropriately. Masks must always be monochrome bitmaps, since they identify which pixels of the primary bitmap are and are not drawn.

Unlike most elements, this element possesses no attributes. However, it does have eight child elements that specify the different pieces of the border. The complete list of child elements is below.

topleft	Exactly one "topleft" child element must appear, specifying the upper left corner.
topright	Exactly one "topright" child element must appear, specifying the upper right corner.
bottomleft	Exactly one "bottomleft" child element must appear, specifying the lower left corner.
bottomright	Exactly one "bottomright" child element must appear, specifying the lower right corner.
left	Exactly one "left" child element must appear, specifying the left edge.
top	Exactly one "top" child element must appear, specifying the top edge.
right	Exactly one "right" child element must appear, specifying the right edge.
bottom	Exactly one "bottom" child element must appear, specifying the bottom edge.

IMPORTANT! The bitmaps comprising a border must be symmetric. This means that six of the bitmaps must possess the same height and another six must possess the same width. Specifically, all bitmaps used across the top and bottom edges must have the same height, which includes: topleft, top, topright, bottomleft, bottom, and bottomright. In addition, all bitmaps used across the left and right edges must have the same width, which includes: topleft, left, bottomleft, topright, right, and bottomright.

Corner Elements

There are four different corners within a border, resulting in four different child elements: "topleft", "topright", "bottomleft", and "bottomright". Each of these elements has the identical set of attributes, so they are all defined here in one place. The complete list of attributes for these elements is below.

bitmap	Text – Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
mask	(Optional) Text – Name of the file containing the monochromatic mask image to be used. This file must have the same dimensions as the bitmap above and must be placed in the same folder where all of the data files for the game system reside. If empty, the bitmap is assumed to be a solid rectangular region with no transparency. Default: Empty.
xoffset	Reserved for future use. Do not specify.
yoffset	Reserved for future use. Do not specify.

Edge Elements

There are four different edges within a border, resulting in four different child elements: "left", "top", "bottom", and "right". Each of these elements has the identical set of attributes, so they are all defined here in one place. The complete list of attributes for these elements is below.

bitmap	Text – Name of the file containing the bitmap image to be used. This file must be placed in the same folder where all of the data files for the game system reside.
mask	(Optional) Text – Name of the file containing the monochromatic mask image to be used. This file must have the same dimensions as the bitmap above and must be placed in the same folder where all of the data files for the game system reside. If empty, the bitmap is assumed to be a solid rectangular region with no transparency. Default: Empty.

Example

The following example demonstrates what various "resource" elements might look like. All default values are assumed for optional attributes.

```
<resource id="color">
  <color color="ff0080"/>
  </resource>

<resource id="font">
  <font face="Arial" size="40" style="bold"/>
  </resource>

<resource id="bitmap">
  <bitmap bitmap="image.bmp"/>
  </resource>

<resource id="solid">
  <solid color="00ff00" thickness="2"/>
  </resource>

<resource id="border">
  <border>
    <topleft bitmap="topleft.bmp"/>
    <topright bitmap="topright.bmp"/>
    <bottomleft bitmap="bottomleft.bmp"/>
    <bottomright bitmap="bottomright.bmp"/>
    <left bitmap="left.bmp"/>
    <top bitmap="top.bmp"/>
    <right bitmap="right.bmp"/>
    <bottom bitmap="bottom.bmp"/>
    </border>
  </resource>
```

Category: Kit Reference

Style Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

Contents

The "style" Element

All of the basic visual look and behaviors of portals is encapsulated in a collection of styles. Each distinct category of portal has its own type of style, and you can only associate styles with portals of the corresponding type. Each separate style is specified through the use of a "style" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the style. This id is used in all references to the style.
border	(Optional) Id – Identifies the border to used in conjunction with this style. All portals may have a border drawn around them, and the border is controlled via the style. You can specify the unique id of the border to use or "none" to indicate no border. Default: "none"

The "style" element also possesses child elements that define the specifics of the style. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! With the exception of the "resource" element, exactly **one** of these child elements may be specified for each style. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of style that is being defined. You may include as many "resource" elements as you wish after the single child element that specifies the style.

style_label	An optional "style_label" element may appear as defined by the given link. This element specifies the details of a style for use with label portals.
style_image	An optional "style_image" element may appear as defined by the given link. This element specifies the details of a style for use with image portals.
style_edit	An optional "style_edit" element may appear as defined by the given link. This element specifies the details of a style for use with edit portals.
style_checkbox	An optional "style_checkbox" element may appear as defined by the given link. This element specifies the details of a style for use with checkbox portals.
style_menu	An optional "style_menu" element may appear as defined by the given link. This element specifies the details of a style for use with menu portals.
style_action	An optional "style_action" element may appear as defined by the given link. This element specifies the details of a style for use with action portals.
style_incrementer	An optional "style_incrementer" element may appear as defined by the given link. This element specifies the details of a style for use with incrementer portals.

style_chooser	An optional "style_chooser" element may appear as defined by the given link. This element specifies the details of a style for use with chooser portals.
style_region	An optional "style_region" element may appear as defined by the given link. This element specifies the details of a style for use with region portals.
style_table	An optional "style_table" element may appear as defined by the given link. This element specifies the details of a style for use with table portals.
style_separator	An optional "style_separator" element may appear as defined by the given link. This element specifies the details of a style for use with separator portals.
style_special	An optional "style_special" element may appear as defined by the given link. This element specifies the details of a style for use with special portals.
style_output	An optional "style_output" element may appear as defined by the given link. This element specifies the details of a style for use with output portals.
resource	Zero or more "resource" elements may appear as defined by the given link. This element specifies new resources that are used in conjunction with the style.

Colors in Styles

Many styles allow you to directly specify a color value in the format "xxxxxx". The format for the color uses standard HTML color syntax, with each character representing a hexadecimal digit. The first two characters define the Red color value, the next two Green, and the last two Blue. For example, the color "ff0080" specifies a Red value of "ff", a Green value of "00", and a Blue value of "80".

For additional details on specifying colors via the HTML syntax, please refer to one of the many websites that provide this information, such as http://www.w3schools.com/Html/html_colors.asp.

Scaling of Images

When encoded text is supported within a category of portal, you will typically be given control over the scaling of images within the corresponding style. When enabled, image scaling is applied to all bitmaps that are inserted into the encoded text that is rendered into the portal. The scaling ratio is based on the difference between the initial font size for the portal and the current font size at which the text is now being rendered. Scaling is valuable for when you want to ensure that the bitmaps remain proportionally sized relative to the font size of the text, which is important when text and bitmaps are interleaved within the encoded text.

The "style_label" Element

The "style_label" element defines the facets of a style for label portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
background	(Optional) Id – Unique id of the bitmap resource to use as the background. If omitted, the text is drawn transparently on the existing background region. Default: Empty.
alignment	(Optional) Set – Specifies how the text should be horizontally aligned within the portal width. Must be one of these values: ▪ left – Text is left-aligned. ▪ center – Text is centered within the portal. ▪ right – Text is right-aligned. ▪ Default: "left".
ispattern	(Optional) Boolean – Indicates whether the background bitmap should be centered within the portal dimensions or tiled to fill the entire portal. Default: "yes".
scaleimage	(Optional) Boolean – Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

The "style_image" Element

The "style_image" element defines the facets of a style for image portals. Since image portals simply contain the image and nothing else, there are no special attributes for this element.

The "style_edit" Element

The "style_edit" element defines the facets of a style for edit portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
backcolor	(Optional) Text – Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
backcolorid	(Optional) Id – Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
alignment	(Optional) Set – Specifies how the text should be horizontally aligned within the portal width. Must be one of these values: ▪ left – Text is left-aligned. ▪ center – Text is centered within the portal. ▪ right – Text is right-aligned. ▪ Default: "left".
autoselect	(Optional) Set – Indicates whether to automatically select the entire text contents of the portal when it gains the focus. Must be one of the following: ▪ yes – Contents are always automatically selected when focus is gained. ▪ no – Contents are never selected when focus is gained. ▪ default – Default handling is performed. If the edit portal contains a numeric field, the contents are automatically selected, but no selection is performed for text-based fields. ▪ Default: "default".
itemborder	(Optional) Id – Unique id of the border resource to draw around each individual edit cell for an "edit_date" portal. If "none", no border is drawn. Default: "none". **NOTE!** An "edit_date" portal can be assigned **either** an item border **or** a border around the entire portal, but never both.
septext	(Optional) Text – Specifies the text to be drawn between each individual edit cell for an "edit_date" portal. Default: "/".
sepfont	(Optional) Id – Unique id of the font resource to be used for drawing the separator text between individual cells for an "edit_date" portal. If empty, the resource specified by the "font" attribute is assumed. Default: Empty.
sepcolor	(Optional) Text – Color value to be used for drawing the separator text in the format "xxxxxx" (see above). If omitted, the "sepcolorid" attribute must be specified. Default: Empty.
sepcolorid	(Optional) Id – Unique id of the color resource to be used for drawing the separator text. If omitted, the "sepcolor" attribute must be specified. Default: Empty.

The "style_checkbox" Element

The "style_checkbox" element defines the facets of a style for checkbox portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.

scaleimage	(Optional) Boolean – Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".
check	(Optional) Id – Unique id of the bitmap resource used to indicate the box is checked. If omitted, a default bitmap it used. Default: Empty.
checkoff	(Optional) Id – Unique id of the bitmap resource used to indicate the box is checked when the portal is disabled. If omitted, a default bitmap it used. Default: Empty.
uncheck	(Optional) Id – Unique id of the bitmap resource used to indicate the box is not checked. If omitted, a default bitmap it used. Default: Empty.
uncheckoff	(Optional) Id – Unique id of the bitmap resource used to indicate the box is not checked when the portal is disabled. If omitted, a default bitmap it used. Default: Empty.

The "style_menu" Element

The "style_menu" element defines the facets of a style for menu portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
backcolor	(Optional) Text – Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
backcolorid	(Optional) Id – Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
selecttext	(Optional) Text – Color value to be used for drawing text of the selected item when the menu is "dropped" in the format "xxxxxx" (see above). If omitted, the "selecttextid" attribute must be specified. Default: Empty.
selecttextid	(Optional) Id – Unique id of the color resource to be used for drawing text of the selected item when the menu is "dropped". If omitted, the "textcolor" attribute must be specified. Default: Empty.
selectback	(Optional) Text – Color value to be used as the background of the selected item when the menu is "dropped" in the format "xxxxxx" (see above). If omitted, the "selectbackid" attribute must be specified. Default: Empty.
selectbackid	(Optional) Id – Unique id of the color resource to be used as the background of the selected item when the menu is "dropped". If omitted, the "backcolor" attribute must be specified. Default: Empty.
activetext	(Optional) Text – Color value to be used for drawing text of the selected item in the format "xxxxxx" (see above). This color is only used within the "non-dropped" region of the menu and allows the color highlighting of invalid menu items without impacting the behavior of the "dropped" menu. If omitted, the "selectbackid" attribute may be specified. If neither color is specified, the "selecttext" color is used. Default: Empty.
activetextid	(Optional) Id – Unique id of the color resource to be used as the background of the selected item. See the "activetext" attribute above for further details. If omitted, the "activetext" attribute may be specified. Default: Empty.
droplist	(Optional) Id – Unique id of the bitmap to be used for the drop arrow on the right when the menu is enabled. If omitted, the system default bitmap is used. Default: Empty.
droplistoff	(Optional) Id – Unique id of the bitmap to be used for the drop arrow on the right when the menu is disabled. If omitted, the system default bitmap is used. Default: Empty.
scaleimage	(Optional) Boolean – Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

The "style_action" Element

The "style_action" element defines the facets of a style for action portals (typically used as clickable buttons). The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute

	must be specified. Default: Empty.
up	Id – Unique id of the bitmap resource to be used as the "up" state of the action portal.
down	Id – Unique id of the bitmap resource to be used as the "down" state of the action portal.
off	Id – Unique id of the bitmap resource to be used as the "up" state of the action portal when the portal is disabled.
xoffset	(Optional) Integer – Specifies the offset adjustment of any text along the horizontal X axis. The text is centered along the axis and can be shifted left or right via this attribute, with positive values shifting to the right and negative values shifting left. Default: "0".
yoffset	(Optional) Integer – Specifies the offset adjustment of any text along the vertical Y axis. The text is centered along the axis and can be shifted up or down via this attribute, with positive values shifting downward and negative values shifting upward. Default: "0".

The "style_incrementer" Element

The "style_incrementer" element defines the facets of a style for incrementer portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
(Optional) background	Id – Unique id of the bitmap resource to be used as the background. If empty, the portal has no background and is drawn transparently. Default: Empty. **NOTE!** If a background bitmap is specified, the incrementer size is dictated by the dimensions of the bitmap. If not specified, then the dimensions must be specified via the "fullwidth" and "fullheight" attributes.
fullwidth	(Optional) Integer – Specifies the fixed width to utilize for all incrementers assigned this style. If "0", a background must be specified to dictate the dimensions. Default: "0".
fullheight	(Optional) Integer – Specifies the fixed height to utilize for all incrementers assigned this style. If "0", a background must be specified to dictate the dimensions. Default: "0".
textleft	Integer – Specifies the left edge of the region in which text is drawn within the incrementer.
texttop	Integer – Specifies the top edge of the region in which text is drawn within the incrementer.
textwidth	Integer – Specifies the width of the region in which text is drawn within the incrementer.
textheight	Integer – Specifies the height of the region in which text is drawn within the incrementer.
plusup	Id – Unique id of the bitmap resource to be used for the "+" button in its "up" state.
plusdown	Id – Unique id of the bitmap resource to be used for the "+" button in its "down" state.
plusoff	Id – Unique id of the bitmap resource to be used for the "+" button in its "up" state when the incrementer is disabled.
plusx	Integer – Specifies the offset along the horizontal X axis where the "+" button is positioned within the overall incrementer portal.
plusy	Integer – Specifies the offset along the vertical Y axis where the "+" button is positioned within the overall incrementer portal.
minusup	Id – Unique id of the bitmap resource to be used for the "-" button in its "up" state.
minusdown	Id – Unique id of the bitmap resource to be used for the "-" button in its "down" state.
minusoff	Id – Unique id of the bitmap resource to be used for the "-" button in its "up" state when the incrementer is disabled.
minusx	Integer – Specifies the offset along the horizontal X axis where the "-" button is positioned within the overall incrementer portal.
minusy	Integer – Specifies the offset along the vertical Y axis where the "-" button is positioned within the overall incrementer portal.
editable	(Optional) Boolean – Indicates whether the incrementer value can be directly user-edited by clicking within the value region. Disabling this can be useful when the value does not correspond to what is displayed, such as when selecting a die type (e.g. d6, d8, d10). Default: "yes".

scaleimage	(Optional) Boolean – Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

The "style_chooser" Element

The "style_chooser" element defines the facets of a style for chooser portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
backcolor	(Optional) Text – Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
backcolorid	(Optional) Id – Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
scaleimage	(Optional) Boolean – Indicates whether bitmaps included within encoded text are scaled (see above). Default: "no".

The "style_region" Element

The "style_region" element defines the facets of a style for region portals. The complete list of attributes for this element is below.

(Optional) background	Id – Unique id of the bitmap resource to be used as the background for the region. If empty, the portal has no background and is drawn transparently. Default: Empty.

The "style_table" Element

The "style_table" element defines the facets of a style for table portals. The complete list of attributes for this element is below.

(Optional) background	Id – Unique id of the bitmap resource to be used as the background for the table. If empty, the portal has no background and is drawn transparently. Default: Empty.
itemborder	(Optional) Id – Unique id of the bitmap resource to be used as a border around each individual item inside the table. If empty, no border is drawn. Default: Empty.
showgridhorz	(Optional) Boolean – Indicates whether horizontal grid lines should be drawn between each item within the table. Default: "no".
showgridvert	(Optional) Boolean – Indicates whether vertical grid lines should be drawn between each item within the table. Default: "no".
gridwidth	(Optional) Integer – Thickness of the grid lines drawn between items within the table (in pixels). The thickness of both horizontal and vertical grid lines is always the same. Default: "1".
gridcolor	(Optional) Text – Color value to be used for drawing grid lines in the format "xxxxxx" (see above). Default: "888888".
gridcolorid	(Optional) Id – Unique id of the color resource to be used for drawing grid lines. If specified, the default value for the "gridcolor" attribute is ignored. Default: Empty.

The "style_separator" Element

The "style_separator" element defines the facets of a style for separator portals. The complete list of attributes for this element is below.

isvertical	(Optional) Boolean – Indicates whether the separator is oriented vertically or horizontally. Default: "no".
start	Id – Unique id of the bitmap resource to be used at the left/top end of the separator.
end	Id – Unique id of the bitmap resource to be used at the right/bottom end of the separator.
center	Id – Unique id of the bitmap resource to be used in the middle of the separator. This bitmap is tiled as necessary to fill the entire span of the separator.

The "style_special" Element

The "style_special" element defines the facets of a style for special portals. Due to its nature, there are no attributes for this element.

The "style_output" Element

The "style_output" element defines the facets of a style for output portals. The complete list of attributes for this element is below.

font	Id – Unique id of the font resource to be used for the style.
textcolor	(Optional) Text – Color value to be used for drawing text in the format "xxxxxx" (see above). If omitted, the "textcolorid" attribute must be specified. Default: Empty.
textcolorid	(Optional) Id – Unique id of the color resource to be used for drawing text. If omitted, the "textcolor" attribute must be specified. Default: Empty.
backcolor	(Optional) Text – Color value to be used as the background in the format "xxxxxx" (see above). If omitted, the "backcolorid" attribute must be specified. Default: Empty.
backcolorid	(Optional) Id – Unique id of the color resource to be used as the background. If omitted, the "backcolor" attribute must be specified. Default: Empty.
alignment	(Optional) Set – Specifies how the text should be horizontally aligned within the portal width (for labels). Must be one of these values: ▪ left – Text is left-aligned. ▪ center – Text is centered within the portal. ▪ right – Text is right-aligned. ▪ Default: "left".
(Optional) background	Id – Unique id of the bitmap resource to be used as the background for tables. If empty, the portal has no background and is drawn transparently. Default: Empty.
itemborder	(Optional) Id – Unique id of the bitmap resource to be used as a border around each individual item inside a table. If empty, no border is drawn. Default: Empty.
showgridhorz	(Optional) Boolean – Indicates whether horizontal grid lines should be drawn between each item within a table. Default: "no".
showgridvert	(Optional) Boolean – Indicates whether vertical grid lines should be drawn between each item within a table. Default: "no".
gridwidth	(Optional) Integer – Thickness of the grid lines drawn between items within a table (in pixels). The thickness of both horizontal and vertical grid lines is always the same. Default: "1".
gridcolor	(Optional) Text – Color value to be used for drawing grid lines in the format "xxxxxx" (see above). Default: "888888".
gridcolorid	(Optional) Id – Unique id of the color resource to be used for drawing grid lines. If specified, the default value for the "gridcolor" attribute is ignored. Default: Empty.

Example

The following example demonstrates what various "style" elements might look like. All default values are assumed for optional attributes.

```
<style id="label">
  <style_label textcolor="f0f0f0" font="fntnormal" alignment="center"/>
  </style>

<style id="edit" border="sunken">
  <style_edit textcolor="d2d2d2" backcolor="000000" font="fntedit" alignment="center"/>
  </style>

<style id="increment">
  <style_incrementer textcolor="f0f0f0" font="fntincrsim"
      textleft="13" texttop="0" textwidth="24" textheight="20"
      fullwidth="50" fullheight="20"
      plusup="incplusup" plusdown="incplusdn" plusoff="incplusof"
      plusx="39" plusy="0"
      minusup="incminusup" minusdown="incminusdn" minusoff="incminusof"
      minusx="0" minusy="0">
      </style_incrementer>
  </style>

<style id="action">
  <style_action textcolor="000088" font="fntactbig"
      up="actbigup" down="actbigdn" off="actbigof">
    </style_action>
  <resource id="actbigup" isbuiltin="yes">
    <bitmap bitmap="button_big_up.bmp"/>
    </resource>
  <resource id="actbigdn" isbuiltin="yes">
    <bitmap bitmap="button_big_down.bmp"/>
    </resource>
  <resource id="actbigof" isbuiltin="yes">
    <bitmap bitmap="button_big_off.bmp"/>
    </resource>
  </style>

<style id="table" border="brdsystem">
  <style_table itemborder="sunken" showgridhorz="yes" gridcolor="808080"/>
  </style>

<style id="checkbox">
  <style_checkbox textcolor="f0f0f0" font="fntcheck"/>
  </style>

<style id="image" border="brdsystem">
  <style_image/>
  </style>

<style id="menu" border="sunken">
  <style_menu font="fntmenu" textcolor="84c8f7" backcolor="2a2c47"
      selecttext="1414f7" selectback="f0f0f0"/>
  </style>

<style id="separator">
  <style_separator isvertical="no"
      start="sephorzsta" center="sephorzmid" end="sephorzend"/>
  <resource id="sephorzsta" isbuiltin="yes">
    <bitmap bitmap="sep_horz_start.bmp"/>
    </resource>
  <resource id="sephorzmid" isbuiltin="yes">
    <bitmap bitmap="sep_horz_middle.bmp"/>
    </resource>
  <resource id="sephorzend" isbuiltin="yes">
    <bitmap bitmap="sep_horz_end.bmp"/>
    </resource>
  </style>

<style id="region" border="brdsystem">
  <style_region/>
  </style>
```

Category: Kit Reference

Component Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

Contents

The "component" Element

At the heart of every thing are the components that define their shared behaviors. All components define a collection of fields and re-usable behaviors that can be combined into component sets. Each component is specified through the use of a "component" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the component. This id is used in all references to the component.
name	Text – Public name associated with the component. Maximum length of 50 characters.
sequence	(Optional) Set – Designates the default sorting sequence to be used for picks based on this component. If no sort set is specified, or if the sort set does not yield a difference, picks will be sorted based on this criteria. Must be one of these value: ▪ ascii – Sort alphabetically based on pick name. ▪ nocase – Ignore case while sorting alphabetically. ▪ id – Sort on the unique id of each pick as if it were text. ▪ Default: "nocase".
nonusersort	(Optional) Set – Designates the special sorting behavior for picks that are **not** user-ordered when the same table mixes picks that are user-order with those that are non-user-ordered. Must be one of the following: ▪ first – Picks that are not user-ordered are sorted before picks that are user-sorted. ▪ last – Picks that are not user-ordered are sorted after picks that are user-sorted. ▪ none – No differentiation is made between user-ordered picks and those that are not user-ordered. ▪ Default: "none". **NOTE!** Any attempt by the user to move non-orderable picks will fail, as will any attempt to move user-ordered picks above or below non-orderable picks.
usernamable	(Optional) Boolean – Indicates whether picks from this component should be user-namable. If a component allows naming, so does any derived component set, unless the component set explicitly says otherwise. Default: "no".
autocompset	(Optional) Boolean – Indicates whether the Kit should automatically create a new component set with the identical unique id as the component. This new compset will possess exactly one component - this one. Default: "yes".

orderfield	(Optional) Id – Unique id to be used when automatically defining a special field for the component that will track the details of user-ordering. If empty, then this component does not support user-ordered sequencing of its picks. Default: Empty.
unwind	(Optional) Id – Unique id of the tag group to utilize for controlling "unwind" logic of picks derived from this component. The unwind mechanism ensures picks are deleted in the reverse order they are added. If omitted, no unwind logic is enabled for the component. Default: Empty.
isgear	(Optional) Boolean – Indicates whether things derived from this component are considered to be "gear". The gear mechanism automatically provides additional logic to all things designated as gear. Default: "no".
denyboot	(Optional) Boolean – Indicates whether things derived from this component are denied the ability to be bootstrapped from any source. Default: "no".
panellink	(Optional) Id – Unique id of an edit panel that is associated with all things derived from this component as the default panel linkage. If a thing also specifies an explicit panel linkage, that will take precedence. If empty, no default panel linkage is established. Default: Empty.
hasshortname	(Optional) Boolean – Indicates whether all compsets derived from this component should automatically incorporate the "short name" behavior provided by the Kit. Default: "no".
ispublic	(Optional) Boolean – Indicates whether this component is displayed within the Editor when the user utilizes the "Find Things" mechanism. In general, internal "helper" components should be made non-public so that the user only interacts with components that will be familiar to them. Default: "yes".

The "component" element also possesses child elements that define various facets of the component. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

field	Zero or more "field" elements may appear as defined by the given link. This element specifies the fields that exist for the component.
usage	Zero or more "usage" elements may appear as defined by the given link. This element specifies the pick-based usage pools associated with all things derived from the component.
linkage	Zero or more "linkage" elements may appear as defined by the given link. This element specifies the pick linkages that can be defined for all things derived from the component.
identity	Zero or more "identity" elements may appear as defined by the given link. This element specifies any identity tag groups that are defined for the component.
containerreq	Zero or more "containerreq" elements may appear as defined by the given link. This element specifies any container requirements for things derived from the component.
displace	An optional "displace" element may appear as defined by the given link. This element specifies any displacement behaviors that the component must perform.
shadow	An optional "shadow" element may appear as defined by the given link. This element specifies any shadowing behaviors that the component must perform.
creation	An optional "creation" element may appear as defined by the given link. This element defines a Creation Script for the component.
deletion	An optional "deletion" element may appear as defined by the given link. This element defines a Deletion Script for the component.
xactsetup	An optional "xactsetup" element may appear as defined by the given link. This element defines an TransactSetup Script for the component.
xactbuy	An optional "xactbuy" element may appear as defined by the given link. This element defines an TransactBuy Script for the component.
xactsell	An optional "xactsell" element may appear as defined by the given link. This element defines an TransactSell Script for the component.
tag	Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to every thing derived from the component.
bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped to every thing derived from the component.
loadfixup	An optional "loadfixup" element may appear as defined by the given link. This element defines a LoadFixup Script for the component.
eval	Zero or more "eval" elements may appear as defined by the given link. This element specifies any Eval Scripts that must be performed for the component.
	Zero or more "evalrule" elements may appear as defined by the given link. This element specifies any EvalRule

evalrule	Scripts that must be performed for the component.
prereq	Zero or more "prereq" elements may appear as defined by the given link. This element specifies any pre-requisite tests that are applied to every thing derived from the component.
merge	An optional "merge" element may appear as defined by the given link. This element defines a Merge Script for the component.
split	An optional "split" element may appear as defined by the given link. This element defines a Split Script for the component.

The "usage" Element

The "usage" element defines a pick-based usage pool to associate with every thing for the component. The complete list of attributes for this element is below.

usage	Id – Unique id of the usage pool to be associated with all picks derived from this component.

The "linkage" Element

The "linkage" element defines a linkage to another pick that is associated with every thing for the component. The complete list of attributes for this element is below.

linkage	Id – Specifies the unique id of the linkage being defined for the component.
optional	(Optional) Boolean – Indicates whether this linkage is optional or required for all picks. Default: "no".

The "identity" Element

The "identity" element defines an identity tag group for the component and all derived things. The complete list of attributes for this element is below.

group	Id – Unique id of the tag group that will be used for identity handling of all things derived from this component. **NOTE!** This tag group does **not** need to exist. If it does not, it is automatically created and setup appropriately by the Kit, but you may not add further dynamic tags to a tag group that is auto-created in this way.

The "displace" Element

The "displace" element defines the displacement behavior for the component and all derived things. The complete list of attributes for this element is below.

target	(Optional) Set – Designates the target into which picks derived from this component will be displaced. Must be one of these values: ▪ hero – Picks are displaced into the hero/actor, regardless of location in the structural hierarchy. ▪ parent – Picks are displaced into the immediate container. ▪ Default: "hero".
PCDATA	TagExpr – Specifies a tag expression that determines whether an individual pick is displaced. This tag expression is applied to the **pick** itself - not the container. Furthermore, the tag expression **does** include tags that are automatically added to the pick at creation, such as auto-tags from a table or a bootstrap. The tag expression is only applied once, when the pick is first added to the container, after which the pick is either displaced or not for the rest of its existence.

The "shadow" Element

The "shadow" element defines the shadowing behavior for the component and all derived things. The complete list of attributes for this element is below.

(Optional) Set – Designates the target into which picks derived from this component will be shadowed. Must be one

target	of these values: ▪ hero – Picks are shadowed into the hero/actor, regardless of location in the structural hierarchy. ▪ parent – Picks are shadowed into the immediate container. ▪ Default: "hero".
PCDATA	TagExpr – Specifies a tag expression that determines whether an individual pick is shadowed. This tag expression is applied to the **pick** itself - not the container. Furthermore, the tag expression **does** include tags that are automatically added to the pick at creation, such as auto-tags from a table or a bootstrap. The tag expression is only applied once, when the pick is first added to the container, after which the pick is either shadowed or not for the rest of its existence.

The "creation" Element

The "creation" element defines a Creation Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Creation script.

The "deletion" Element

The "deletion" element defines a Deletion Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Deletion script.

The "xactsetup" Element

The "xactsetup" element defines an TransactSetup Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the TransactSetup script.

The "xactbuy" Element

The "xactbuy" element defines an TransactBuy Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the TransactBuy script.

The "xactsell" Element

The "xactsell" element defines an TransactSell Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the TransactSell script.

The "loadfixup" Element

The "loadfixup" element defines a LoadFixup Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the LoadFixup script.

The "merge" Element

The "merge" element defines a Merge Script for the component. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Merge script.

The "split" Element

The "split" element defines a Split Script for the component. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the Split script.

Example

The following example demonstrates what a "component" element might look like. All default values are assumed for optional attributes.

```
<component id="WeaponBase" name="Weapon"
      autocompset="no" hasshortname="yes" panellink="armory">
  <field id="wpDamage" name="Damage" type="derived" maxlength="20"/>
  <field id="wpNetAtk" name="Net Attack" type="derived"/>
  <usage usagepool="mypool"/>
  <linkage linkage="attribute" optional="no"/>
  <identity group="Weapon"/>
  <displace target="hero">Helper.Displace</displace>
  <tag group="Equipment" tag="Hand"/>
  <eval index="1" phase="Final" priority="5000"><![CDATA[
    ~insert script code here
    ]]></eval>
  <evalrule value="1" phase="Validate" priority="5000" message="Resource overspent">
    ~insert script code here
    ]]></evalrule>
  <prereq message="Requirement not met">
    <match>Helper.WeapCheck</match>
    <valid><![CDATA[
      ~insert script code here
      ]]></valid>
    </prereq>
  </component>
```

Category: Kit Reference

Field Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference … Component Element (Data)

Contents

The "field" Element

Most components will possess at least one field. These fields represents values associated with all things derived from the component, such as attack and damage values for weapons, ranges and components for spells, etc. Extensive details on the use of fields can be found in the separate section on Advanced Fields. Each separate field is specified through the use of a "field" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id to utilize for this field.
name	Text – Specifies the publicly visible name associated with the field. Maximum length is 30 characters.
abbrev	(Optional) Text – Specifies a shortened abbreviation to use for the field name. If empty, the name is used as the abbreviation and truncated as necessary. Maximum length is 10 characters. Default: Empty.
maxlength	(Optional) Integer – Specifies the maximum number of characters that will be managed for this field. If zero, the field is considered a value-based field. Default: "0".
type	(Optional) Set – Designates the type of access behavior granted by this field. Field values for picks always start out equal to the value assigned to the thing, and they can often be changed for the pick during evaluation. Must be one of these values: static – The value assigned to the thing is fixed and can never be changed. user – The value is user-configurable via some appropriate method, such as an edit portal for text-based fields. derived – The value is calculated during the evaluation cycle. Default: "user".
style	(Optional) Set – Designates any special characteristics of the field. Must be one of these values: normal – The field is used normally as a simple value or string. menu – The field contains an item selected via a menu. array – The field encompasses an array of values, with the size specified via the "arrayrows" attribute. matrix – The field encompasses a matrix of values, with the size specified via both the "arrayrows" and "maxtrixcolumns" attributes. Default: "normal".
decimals	(Optional) Integer – Specifies the number of decimal places to be included when outputting a value-based field as text. Values are always tracked internally as floating point numbers, but this attribute governs how the field is displayed as text. If zero, the field is always output as an integer value. Default: "0".
defvalue	(Optional) Text – Specifies the default value to be assigned for this field on every thing that possesses the field. If the field is value-based, then this attribute is assumed to be value-based as well and is converted to a value for assignment, with the empty string being treated as zero. Default: "".
arrayrows	(Optional) Integer – Specifies the number of rows possessed by a field designated as either an array or matrix. Default: "1".
matrixcolumns	(Optional) Integer – Specifies the number of columns possessed by a field designated as a matrix. Default: "1".
minvalue	(Optional) Integer – Specifies the minimum value that the field can assume. If a value lower than this is ever assigned to the field, it is bounded to this value. Default: "-999999999999999.".
maxvalue	(Optional) Integer – Specifies the maximum value that the field can assume. If a value greater than this is ever assigned to the field, it is bounded to this value. Default: "999999999999999.".
maxfinal	(Optional) Integer – Specifies the maximum number of characters that any finalized value may possess. If zero, no finalization is performed for the field. If non-zero, a Finalize script must be provided. Default: "0".

	NOTE! Finalization is only supported for value-based fields.
nevercache	(Optional) Boolean – Indicates that the results of finalization for this field should **never** be cached and always regenerated. This attribute applies only to finalization of values on **picks**. Default: "no".
iscachething	(Optional) Boolean – Indicates whether the finalized results for this field should be cached for **things**. Finalized values are rarely utilized for things, but they are expensive to re-calculate when used on things, so this attribute allows things to cache their finalized values. Default: "no".
persistence	(Optional) Set – Designates the persistence behavior for this field, which means how the value is managed across evaluation cycles and whether it is saved/loaded to/from portfolios. Persistence is not applicable for "static" fields and is always performed for "user" fields, so this attribute only applies to "derived" fields. Must be one of these values: none – The field has no special persistence behavior, being reset at the start of every evaluation cycle and never saved in portfolios. noreset – The field is never reset at the start of evaluation cycles and never saved in portfolios. full – The field is never reset and is saved in portfolios for restoration when reloaded. Default: "none"
usedelta	(Optional) Boolean – Indicates whether a separate "delta" value should be maintained for the field. Deltas make it possible to easily have users edit values that are intuitive, even though they integrate adjustments that are applied from other effects. The delta is only applicable to "user" fields that are value-based. Default: "no".
ismultiline	(Optional) Boolean – Indicates whether the field should always be considered to contain multi-line text. If a field is multi-line, all portals mapped to the field are implicitly designated as multi-line. If a field is not multi-line, then portals can make their own determination regarding multi-line behavior. This attribute only applies to text-based fields. Default: "no".
signed	(Optional) Boolean – Indicates whether the field should always be considered to contain a signed value. If field is signed, then any portal mapped to the field is implicitly designated as signed, else portals are free to make their own determination regarding signed behavior. Default: "no".
format	(Optional) Set – Designates any special numeric formatting to be enforced for a value-based field. Must be one of these values: integer – Any portal mapped to the field will always treat the field as an integer value. float – Any portal mapped to the field will always treat the field as a floating-point value. any – Portals are free to determine how to present the field for editing. Default: "any".
history	(Optional) Set – Designates what type of history tracking is to be performed for this field. Must be one of these values: none – No history tracking is performed. stack – Tracks all changes in the order applied. changes – Tracks all changes in the order applied, ignoring any adjustments that yield no actual change (e.g. "+0" or "*1"). best – Records only the single largest adjustment, and all changes must use the same modify operator. If an adjustment results in no actual change, it is ignored. Default: "none". **NOTE!** History tracking is only supported for fields that meet the following criteria: type must be "derived"; style must be "normal"; must be value-based; may not be persistent; may not possess a Calculate or Bound script.

The "field" element also possesses child elements that pertain to the individual field value. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

bound	An optional "bound" element may appear as defined by the given link. This element defines the Bound Script that sets up the bounding limits to be used for the field value. If omitted, bounding is performed against the "minvalue" and "maxvalue" attributes. **NOTE!** A Bound script may only be used on a value-based field.
calculate	An optional "calculate" element may appear as defined by the given link. This element defines the Calculate Script that computes the value to be utilized for the field. If omitted, the field value is only modified via eval scripts. **NOTE!** A Calculate script may only be used on a "derived" field.
finalize	An optional "finalize" element may appear as defined by the given link. This element defines the Finalize Script that synthesizes a final value for display to the user. If omitted, no final formatting is performed.

NOTE! A Finalize script may only be used on a value-based field.

The "bound" Element

The "bound" element defines the Bound script that dynamically calculates the bounding limits for the field value. The complete list of attributes for this element is below.

phase	Id – Specifies the unique id of the evaluation phase during which the script is invoked.
priority	Integer – Specifies the evaluation priority during which the script is invoked.
name	(Optional) Text – Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
isprimary	(Optional) Boolean – Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
PCDATA	Script – Specifies the code comprising the Bound script.

The "calculate" Element

The "calculate" element defines the Calculate script that dynamically determines the field value. The complete list of attributes for this element is below.

phase	Id – Specifies the unique id of the evaluation phase during which the script is invoked.
priority	Integer – Specifies the evaluation priority during which the script is invoked.
name	(Optional) Text – Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
isprimary	(Optional) Boolean – Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
PCDATA	Script – Specifies the code comprising the Bound script.

The "finalize" Element

The "finalize" element defines the Finalize script that synthesizes the final value for display to the user. The synthesized value may not exceed the maximum length specified by the "maxfinal" attribute for the field. The complete list of attributes for this element is below.

phase	Id – Specifies the unique id of the evaluation phase during which the script is invoked.
priority	Integer – Specifies the evaluation priority during which the script is invoked.
name	(Optional) Text – Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
isprimary	(Optional) Boolean – Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
PCDATA	Script – Specifies the code comprising the Bound script.

Example

The following example demonstrates what a "field" element might look like. All default values are assumed for optional attributes.

```
<field
  id="wpDamage"
  name="Damage"
  type="derived"
  maxlength="20">
  </field>

<field
  id="perHeight"
  name="Height"
  type="user"
  maxfinal="20"
  defvalue="68">
  <bound phase="Render" priority="10000"><![CDATA[
    @minimum = field[perHtMin].value
    @maximum = field[perHtMax].value
    ]]></bound>
  <finalize><![CDATA[
    ~calculate the height in terms of feet and inches
    var feet as number
    var inches as number
    feet = @value / 12
    feet = round(feet,0,-1)
    inches = @value - (feet * 12)
    ~synthesize appropriate text to display the height properly
    @text = feet & "'"
    if (inches <> 0) then
      @text = @text & " " & inches & chr(34)
      endif
    ]]></finalize>
  </field>
```

Category: Kit Reference

ContainerReq Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

The "containerreq" Element

There will be times when you need to establish a condition for a thing where its container must satisfy a set of criteria. The criteria are spelled out via a Container tag expression. The tag expression is applied against the prospective container of the thing. If the container does not satisfy the tag expression, then the thing must be treated as if it simply doesn't exist within the container. Each separate set of requirements is specified through the use of a "containerreq" element. The complete list of attributes for this element is below.

phase	Id – Specifies the unique id of the evaluation phase during which the tag expression is tested.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested.
name	(Optional) Text – Specifies the name assigned to this test for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this test. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Container tag expression.

The "containerreq" element also possesses child elements that pertain to the requirements upon the container. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines the Match Tag Expression. If omitted, all things are assumed to match and the container requirement test is applied to them all. IMPORTANT! This element is only applicable when the container requirement is defined within a component. In all other cases, this element may not be specified.
before	Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.
after	Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the container requirement.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing is subject to the container requirement. The tag expression is applied against each thing derived from the component, and the requirement is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Match tag expression.

Example

The following example demonstrates what a "containerreq" element might look like. All default values are assumed for optional attributes.

```
<containerreq phase="Setup" priority="500" name="MyTest>
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
  val:Level.? >= 4
  </containerreq>
```

Category: Kit Reference

Tag Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

The "tag" Element

When you wish to assign a tag to a component or thing, you will utilize a "tag" element. The complete list of attributes for this element is below.

group	Id – Specifies the unique id of the tag group to which the tag being assigned belongs.
tag	Id – Specifies the unique id of the tag that must be assigned.
name	(Optional) Text – Specifies the name to be used for the tag. If empty, the unique id, as a text string, is used as the tag's name. Maximum length is 100 characters. Default: Empty.
abbrev	(Optional) Text – Specifies the abbreviation to be used for the tag. If empty, the name is used as the abbreviation, subject to any necessary truncation. Maximum length is 100 characters. Default: Empty.

IMPORTANT! If the tag group is dynamic, you can define new tags by simply assigning them to a component or a thing. When that occurs, you should always specify an appropriate name and abbreviation for the tag. The compiler defines new tags as they are encountered, so there is no way to ensure that one use of a new tag occurs before another, except for the basic rules governing the order in which data files with different file extensions are loaded.

Example

The following example demonstrates what a "tag" element might look like. All default values are assumed for optional attributes.

```
<tag group="MyGroup" tag="MyTag" name="Tag Name" abbrev="Abbrev"/>
```

Category: Kit Reference

Eval Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

The "eval" Element

A major facet of data file authoring it defining scripts on components and things for execution during the evaluation cycle. These scripts are referred to as Eval scripts and are invoked on each individual pick within the actor. Each separate script is specified through the use of an "eval" element. The complete list of attributes for this element is below.

NOTE! For more information on many of these attributes, please see Advanced Script Handling.

phase	Id – Specifies the unique id of the evaluation phase during which the script is invoked.
priority	Integer – Specifies the evaluation priority during which the script is invoked.
index	(Optional) Integer – Assigns an arbitrary, but unique, value to this script. This index is used within error messages to identify the script where the problem occurs. If the script is assigned a name (below), no index is necessary. Default: "1".
runlimit	(Optional) Integer – Specifies the maximum number of times this script will be invoked during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". **NOTE!** The limit is imposed separately within each container, so the script will always be invoked for picks within different containers.
iseach	(Optional) Boolean – Indicates whether the "runlimit" applies individually to each thing or collectively to all things derived from the component. This attribute is only applicable component scripts. Default: "yes".
sortas	(Optional) Id – Specifies the unique id of a different component for which the script will be sorted when establishing the task evaluation sequence. This attribute is only applicable to component scripts. If empty, the script uses the sequencing for the component it is defined within. Default: Empty.
name	(Optional) Text – Specifies the name assigned to this script for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this script. Default: Empty.
isprimary	(Optional) Boolean – Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
PCDATA	Script – Specifies the code comprising the Eval script.

The "eval" element also possesses child elements that pertain to the handling of the script. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the script to be assigned to each thing. If omitted, the script is applied to all derived things. IMPORTANT! This element is only applicable when the condition test is defined within a component. In all other cases, this element may not be specified.
before	Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the script.
after	Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the script.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the eval script. The tag expression is applied against each thing derived from the component, and the script is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Match tag expression.

Example

The following example demonstrates what an "eval" element might look like. All default values are assumed for optional attributes.

```
<eval index="1" phase="Setup" priority="500">
```

```
   <before name="BeforeTest"/>
   <after name="AfterTest"/>
   debug "Script Executing"
   </eval>
```

Category: Kit Reference

EvalRule Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

The "evalrule" Element

Throughout your data files, you'll want to verify facets of different picks within actors. This is accomplished by writing rule scripts on components and things for execution during the evaluation cycle. These rule scripts are referred to as EvalRule scripts, or often simply as "eval rules", and are invoked on each individual pick within the actor. Each separate rule is specified through the use of an "evalrule" element. The complete list of attributes for this element is below.

NOTE! For more information on many of these attributes, please see Advanced Script Handling.

phase	Id – Specifies the unique id of the evaluation phase during which the rule is invoked.
priority	Integer – Specifies the evaluation priority during which the rule is invoked.
message	Text – Specifies the error message displayed to the user within the validation report when the rule is not satisfied.
summary	(Optional) Text – Specifies the summary message displayed within the validation summary bar at the bottom of the main window when the rule is not satisfied. If empty, the message is used as the summary. Default: Empty.
index	(Optional) Integer – Assigns an arbitrary, but unique, value to this rule. This index is used within error messages to identify the rule where the problem occurs. If the rule is assigned a name (below), no index is necessary. Default: "1".
severity	(Optional) Set – Identifies the severity level associated with failing the rule. Must be one of these values: ▪ error - The rule indicates a serious error within the character ▪ warning - The rule indicates a non-critical warning within the character ▪ Default: "error".
runlimit	(Optional) Integer – Specifies the maximum number of times this rule will be invoked during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". **NOTE!** The limit is imposed separately within each container, so the rule will always be invoked for picks within different containers.
iseach	(Optional) Boolean – Indicates whether the "runlimit" applies individually to each thing or collectively to all things derived from the component. This attribute is only applicable component scripts. Default: "yes".
reportlimit	(Optional) Integer – Specifies the maximum number of times the message for this rule will be reported during the evaluation cycle for each container. A value of zero indicates no limit. Default: "0". **NOTE!** The limit is imposed separately within each container, so failures of the rule will always be reported for picks within different containers.
issilent	(Optional) Boolean – Indicates whether the rule should report an error message to the user if failed. In some cases, you'll simply want to properly highlight picks as invalid within the interface without reporting individual errors. In such cases, you can have the rule test all picks and mark them as invalid, but suppress any actual message. Default: "no".
sortas	(Optional) Id – Specifies the unique id of a different component for which the rule will be sorted when establishing the task evaluation sequence. This attribute is only applicable to component rules. If empty, the rule uses the sequencing for the component it is defined within. Default: Empty.
name	(Optional) Text – Specifies the name assigned to this rule for the purpose of establishing timing dependencies. If empty, no timing dependencies may be defined elsewhere that depend upon this rule. Default: Empty.
isprimary	(Optional) Boolean – Indicates whether this task should be considered the "primary" task when multiple tasks are assigned the same name. Default: "no".
PCDATA	Script – Specifies the code comprising the EvalRule script.

The "evalrule" element also possesses child elements that pertain to the handling of the rule. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the rule to be assigned to each thing. If omitted, the rule is applied to all derived things. IMPORTANT! This element is only applicable when the condition test is defined within a component. In all other cases,

this element may not be specified.

before	Zero or more "before" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the rule.
after	Zero or more "after" elements may appear as defined by the given link. This element specifies appropriate timing dependencies possessed by the rule.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the eval rule. The tag expression is applied against each thing derived from the component, and the rule is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA TagExpr – Specifies the code comprising the Match tag expression.

Example

The following example demonstrates what an "evalrule" element might look like. All default values are assumed for optional attributes.

```
<evalrule index="1" phase="Setup" priority="500" message="Rule Failed">
  <before name="BeforeTest"/>
  <after name="AfterTest"/>
  @valid = 1
  </evalrule>
```

Category: Kit Reference

PreReq Element (Data)

Context: HL Kit … Kit Reference … Multiple Sources

Contents

The "prereq" Element

You can establish dependencies wherein certain things require specific conditions to be satisfied by their prospective container in order to be added. These conditions are referred to as Pre-Requisites and are always tested against the prospective container for a thing. Each pre-requisite is specified through the use of a "prereq" element. The complete list of attributes for this element is below.

message	Text – Specifies the error message displayed to the user within the validation report when the pre-requisite is not satisfied.
iserror	(Optional) Boolean – Indicates whether the failing the pre-requisite is considered to an error or merely a warning. This only applies if the user chooses to ignore a failed pre-requisite and add the thing to the container anyways. Default: "yes".
onlyonce	(Optional) Boolean – Indicates whether the pre-requisite should only be reported to the user a single time if it fails. This is important for situations where the pre-requisite is assigned to a thing that is added to a container multiple times, such as class levels within the d20 System. This attribute is only applicable to pre-requisites specified directly on individual things. Default: "no".
issilent	(Optional) Boolean – Indicates whether the pre-requisite should report an error message to the user if failed. In some cases, you'll simply want to properly highlight picks as invalid within the interface without reporting individual errors. In such cases, you can have the pre-requisite perform its tests and mark picks as invalid, but suppress any actual message. Default: "no".

The "prereq" element also possesses child elements that pertain to the handling of the requirement. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

match	An optional "match" element may appear as defined by the given link. This element defines a Match Tag Expression that must be satisfied in order for the pre-requisite to be assigned to each thing. If omitted, the pre-requisite is applied to all derived things. IMPORTANT! This element is only applicable when the pre-requisite is defined within a component. In all other cases, this element may not be specified.
test	An optional "test" element may appear as defined by the given link. This element defines a Container Tag Expression that determines whether the pre-requisite is satisfied. If omitted, the pre-requisite is assumed to satisfy this test.
validate	An optional "valid" element may appear as defined by the given link. This element defines a Validate Script that determines whether the pre-requisite is satisfied. If omitted, the pre-requisite is assumed to satisfy this test.

The "match" Element

The "match" element defines the Match tag expression that determines whether a particular thing is assigned the pre-requisite. The tag expression is applied against each thing derived from the component, and the pre-requisite is only assigned to things that satisfy the tag expression. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Match tag expression.

The "test" Element

The "test" element defines the Container tag expression that determines whether the pre-requisite is satisfied. The tag expression is applied against all the tags of the container, and the pre-requisite is considered valid if the tag expression is satisfied. The complete list of attributes for this element is below.

PCDATA TagExpr – Specifies the code comprising the Container tag expression.

The "validate" Element

The "validate" element defines the Validate script that determines whether the pre-requisite is satisfied. The script is applied against the container, and the pre-requisite is considered valid if the script reports the container as valid. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the Validate script.

Example

The following example demonstrates what an "prereq" element might look like. All default values are assumed for optional attributes.

```
<prereq message="Requirement Failed.">
  <valid>
    @valid = 1
    </valid>
  </prereq>
```

Category: Kit Reference

Component Set Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

Contents

The "compset" Element

Every "thing" you'll define throughout the data files will be based on a specific component set (or compset). Each component set is defined through the use of a "compset" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the compset. This id is used in all references to the compset.
usernamable	(Optional) Set – Designates whether the user is allowed to rename picks that are based on this compset. Must be one of these values: ▪ yes – The user is always allowed to rename picks derived from this compset. ▪ no – The user is never allowed to rename picks derived from this compset. ▪ default – Whether the user can rename picks derived from this compset is dictated by the components upon which the compset is based. If **any** component allows renaming, so does the compset. ▪ Default: "default".
stackable	(Optional) Boolean – Indicates whether picks derived from this compset enable stacking behavior. Default: "no".
forceunique	(Optional) Set – Specifies whether all things derived from this compset must be designated as unique or non-unique. Must be one of these values: ▪ yes – All things derived from this compset must be designated as unique. ▪ no – No things derived from this compset may be designated as unique. ▪ default – Author is free to designate a mixture of unique and non-unique things based on this compset. ▪ Default: "default". **NOTE!** If uniqueness is forced either direction, the compiler will report an error for any thing that is designated incorrectly.

The "compset" element also possesses child elements that pertain to the handling of its components. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

compref	One or more "compref" elements must appear as defined by the given link. This element specifies the individual components that comprise the compset.
distinct	Zero or more "distinct" elements may appear as defined by the given link. This element identifies additional criteria for determining whether two things can be stacked.

The "compref" Element

The "compref" element identifies each of the components that the compset will be built upon. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component to be included in the compset.

The "distinct" Element

The "distinct" element identifies specially treated fields within the compset (i.e. its components). These fields are used to uniquely identify picks and potentially preclude whether those picks can be stacked with other picks. If two picks have the same values for the specified fields, their stackable behavior is dictated by the normal rules for stackability. However, if any "distinct" field differs, the two picks are considered to be **not** stackable. The complete list of attributes for this element is below.

NOTE! If no "distinct" fields are specified, normal stacking rules apply. If one or more "distinct" fields are given, all components within the compset must be stackable. Array-based and matrix-based fields cannot be designated as "distinct".

field Id – Specifies the unique id of the field to be verified as distinct before stacking is allowed.

Example

The following example demonstrates what an "compset" element might look like. All default values are assumed for optional attributes.

```
<compset id="Melee" stackable="yes">
  <compref component="WeaponBase"/>
  <compref component="WeapMelee"/>
  <compref component="Equippable"/>
  <compref component="Gear"/>
  </compset>
```

Category: Kit Reference

Entity Element (Data)

Context: HL Kit … Kit Reference … Structural File Reference

The "entity" Element

There will be times when you need to create a separate container for picks that is distinct from an actor. For example, a customizable weapon or vehicle can have its own set of picks that tailor that specific object. These objects are referred to as entities and gizmos. Each entity is defined through the use of a "entity" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the entity. This id is used in all references to the entity.
form	(Optional) Id – Specifies the unique id of the form that will be used to edit the contents of all gizmos based on this entity. If empty, there is no ability for the user to directly edit the contents of the gizmo. Default: Empty.
defaultthing	(Optional) Id – Specifies the unique id of a thing to be used as the "default" within the context of the entity. All tables assume a "default" thing of the "actor" pick. However, if you define tables for manipulating picks within a gizmo, attempts to use the "actor" pick will fail. This attribute allows you to specify an alternate pick to be used as the default and must designate a pick that always exists within the gizmo (i.e. a thing that is bootstrapped into it). If empty, no suitable default is setup, but you won't always need one. Default: Empty.

The "entity" element also possesses child elements that pertain to its handling. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into every gizmo derived from the entity.
integrity	An optional "integrity" element may appear as defined by the given link. This element defines an Integrity Script for the entity.

The "integrity" Element

The "integrity" element defines an Integrity Script for the entity that imposes rules for the ensuring the user can only save changes to a valid gizmo. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Integrity script.

Example

The following example demonstrates what an "entity" element might look like. All default values are assumed for optional attributes.

```
<entity id="MyEntity" panel="EntityForm" defaultthing="EntityHelp">
  <bootstrap thing="EntityHelp"/>
  <bootstrap thing="EntityInfo"/>
  </entity>
```

Category: Kit Reference

Sort Set Element (Data)

Context: HL Kit ... Kit Reference ... Structural File Reference

The "sortset" Element

The mechanism used for controlling the sorting sequences of things and picks within the Kit is referred to as "sort sets". Each sort set is defined through the use of a "sortset" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the sort set. This id is used in all references to the sort set.
name	Text – Specifies the public name to be used for the sort set.

The "sortset" element also possesses child elements that pertain to its handling. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

sortkey	One or more "sortkey" elements must appear as defined by the given link. This element specifies each of the sort criteria to used, and the order in which the sort key are defined dictates the sequence in which they will be applied. Consequently, the first sort key represents the primary comparison test used, the second sort key is the secondary comparison, etc. The comparisons are applied in order until the objects are differentiated.

The "sortkey" Element

The "sortkey" element defines the particulars for an individual comparison test that will be employed when sorting. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the field or tag group to be used for sorting.
isfield	(Optional) Boolean – Indicates whether this sort key will compare objects based on a field value or the presence of tags. It also identifies the nature of the "id" attribute, which will be the unique id of either a field or tag group, as appropriate. Default: "no".
isascend	(Optional) Boolean – Indicates whether the sort key will sequence objects in ascending or descending order. Default: "yes".

Example

The following example demonstrates what a "sortset" element might look like. All default values are assumed for optional attributes.

```
<sortset id="Armory" name="Weapons and Armor">
  <sortkey isfield="no" id="Equipped"/>
  <sortkey isfield="no" id="Armory"/>
  <sortkey isfield="no" id="_Name_"/>
  </sortset>
```

Category: Kit Reference

Data File Reference

Context: HL Kit … Kit Reference

Overview

The data file is where you'll be defining all of the top-level elements that the user will directly control. This includes visual elements like panels, layouts, templates, and portals. It also includes game system elements like things.

All data files have the ".dat" file extension and are loaded after all of the structural files have defined the framework for the game system.

This section outlines the structure and mechanics for writing a definition file.

IMPORTANT! This section utilizes critical notational conventions that should be reviewed.

IMPORTANT! Just because you **can** put numerous different elements in the same file does not mean you **should** do so. Keeping your data files small and focused will also keep them much more manageable, so break up all the information across files where appropriate. See the Skeleton data files for examples of this.

Structural Composition

The overall file structure is that of a standard XML file. The file must start with an XML version element in the form: "<?xml version="1.0"?>". Following this, the top-level XML element must be a "document" and it must have a "signature" attribute containing the explicit value "Hero Lab Data".

The following table defines the attributes for a "document" element.

signature	Text – Must be the value "Hero Lab Data".

Within the document element, every data file possesses the following child elements, appearing in the sequence given and with the names specified.

procedure	Zero or more "procedure" elements may appear as defined by the given link. This element specifies a collection of procedures that may be called by scripts throughout the data files.
thing	Zero or more "thing" elements may appear as defined by the given link. This element specifies various thing objects for use within the game system.
portal	Zero or more "portal" elements may appear as defined by the given link. This element specifies an assortment of portals for use within layouts.
template	Zero or more "template" elements may appear as defined by the given link. This element specifies an assortment of templates for use within layouts.
layout	Zero or more "layout" elements may appear as defined by the given link. This element specifies an assortment of layouts for use within panels, forms, and sheets.
panel	Zero or more "panel" elements may appear as defined by the given link. This element specifies a variety of panels that will be presented to the user as part of the game system.
form	Zero or more "form" elements may appear as defined by the given link. This element specifies a variety of forms that will be presented to the user as part of the game system.
sheet	Zero or more "sheet" elements may appear as defined by the given link. This element specifies a variety of sheets that will be used for printouts within dossiers.
dossier	Zero or more "dossier" elements may appear as defined by the given link. This element specifies a collection of dossiers that the user can select for character output.
hidden	Zero or more "hidden" elements may appear as defined by the given link. This element identifies things that HL should remove from use within the game system.
editthing	Zero or more "editthing" elements may appear as defined by the given link. This element specifies a variety of elements that enable use of the integrated Editor for creation of things.
faq	Zero or more "faq" elements may appear as defined by the given link. This element specifies various FAQ entries that will be included within the FAQ page for the game system.

Category: Kit Reference

Procedure Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

The "procedure" Element

If you need to invoke the same script code from multiple scripts, then you should consider writing a re-usable procedure that defines the code once and can be called from multiple scripts. Each procedure is defined through the use of a "procedure" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the procedure. This id is used in all references to the procedure.
context	(Optional) Set – Designates a general script context from which this procedure is designed to be called. A script context spans multiple different script types that are related in nature and behavior. Must be one of these values: ▪ unknown – Procedure may only be invoked from within a specific type of script, as dictated by the "scripttype" attribute (below). ▪ pick – Procedure can be called from any script with a pick as its initial context. ▪ container – Procedure can be called from any script with a container as its initial context. ▪ entity – Procedure can be called from entity-related scripts, such as the TitleBar Script. ▪ info – Procedure can be called from any information synthesis script for a thing or pick, such as the MouseInfo Script. ▪ transact – Procedure can be called from any transaction script, such as the TransactBuy Script. ▪ combat – Procedure can be called from any combat-related script, such as the NewCombat Script. ▪ Default: "unknown".
scripttype	(Optional) Set – Designates the specific script type from which this procedure may be called. The procedure may only be called from scripts of this type. Must be one of these values: ▪ unknown – Procedure may be invoked from a general category of scripts, as dictated by the "context" attribute (above). ▪ finalize – Procedure may only be called from within a Finalize Script. ▪ calculate – Procedure may only be called from within a Calculate Script. ▪ bounds – Procedure may only be called from within a Bound Script. ▪ eval – Procedure may only be called from within an Eval Script. ▪ evalrule – Procedure may only be called from within an EvalRule Script. ▪ mouseinfo – Procedure may only be called from within a MouseInfo Script. ▪ titlebar – Procedure may only be called from within a TitleBar Script. ▪ description – Procedure may only be called from within a Description Script. ▪ trigger – Procedure may only be called from within a Trigger Script. ▪ label – Procedure may only be called from within a Label Script. ▪ validate – Procedure may only be called from within a Validate Script. ▪ xactsetup – Procedure may only be called from within a TransactSetup Script. ▪ xactbuy – Procedure may only be called from within a TransactBuy Script. ▪ xactsell – Procedure may only be called from within a TransactSell Script. ▪ newcombat – Procedure may only be called from within a NewCombat Script. ▪ endcombat – Procedure may only be called from within an EndCombat Script. ▪ newturn – Procedure may only be called from within a NewTurn Script. ▪ integrate – Procedure may only be called from within an Integrate Script. ▪ synthesize – Procedure may only be called from within a Synthesize Script. ▪ none – Procedure may be called from **any** type of script. However, there is no initial context for identifiers, so no context transitions may be specified that don't explicitly designate a safe initial context (e.g. "hero."). This procedure type is useful when all inputs can be passed via variables and all results returned via variables.

-
- Default: "unknown".

NOTE! It is **valid** to setup a focus pick via "setfocus" within a calling script and then utilize the inherited focus pick within a procedure of type "none".

PCDATA	Script – Specifies the code comprising the procedure.

Example

The following example demonstrates what a "procedure" element might look like. All default values are assumed for optional attributes.

```
<procedure id="MyProc" context="info">
  ~insert script code here
  </procedure>

<procedure id="MyProc" scripttype="eval">
  ~insert script code here
  </procedure>
```

Category: Kit Reference

Thing Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

Contents

The "thing" Element

The vast majority of objects that will comprise your data files are "things", which will be added to actors and customized via scripts. By definition, all behaviors for things are inherited by any derived picks, unless specified otherwise. Each thing is specified through the use of a "thing" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the thing. This id is used in all references to the thing.
compset	Id – Specifies the unique id of the component set from which this thing is derived.
name	Text – Public name associated with the thing. Maximum length of 100 characters.
description	(Optional) Text – Detailed description text associated with the thing, for display to the user. Default: Empty.
summary	(Optional) Text – Brief summary description for the thing, for display to the user in space-constrained situations. If empty, the full description is always used as the summary. Default: Empty.
isunique	(Optional) Boolean – Indicates whether this thing is treated as unique within each container. If unique, a single pick is only ever added, while non-unique things can be separately added any number of times. Default: "no".
holdable	(Optional) Boolean – Indicates whether this thing can be held within other other things, such as backpacks hold various pieces of gear. Default: "yes".
maxlimit	(Optional) Integer – Specifies the maximum number of instances of this thing that can be added to a given container. If zero, there is no limit imposed. Default: "0".
panellink	(Optional) Id – Specifies the unique id of a panel that is officially associated with this thing. Scripts can generically determine the panel linked to a thing to mark it invalid if picks within the panel are invalid. If empty, the default panel linkage defined by components is assumed. Default: Empty.
stacking	(Optional) Set – Designates the default stacking behavior to be assumed whenever this thing is purchased by the user. Must be one of these values: ▪ solo – Item is created individually, so buying 5 of the thing will add 5 separate instances of the thing to the container. ▪ new – Item is purchased as a new group, so buying 5 of the thing will add one new instance of the thing that has a quantity of 5. ▪ merge – Item is merged to any existing instance of the thing within the container, adding the quantity purchased to the existing quantity. If item does not currently exist, "new" behavior is used. ▪ never – Item can never support stacking. ▪ Default: "solo".
lotsize	(Optional) Integer – Specifies the number of items typically purchased as a single unit when buying this thing. For example, bullets might be purchased by the box, with a box having 25 bullets in it. Default: "1".
	(Optional) Id – Specifies the unique id of another thing which is completely replaced by this thing. All references

replaces	to the replaced thing are automatically mapped to this thing, such as bootstrapping. This allows you to wholesale replace a built-in item with new behaviors of your own choosing. If empty, no replacement behavior is utilized. Default: Empty.
buytemplate	(Optional) Id – Specifies the unique id of a template that is used for purchasing the thing. This attribute is only applicable to things which have a child entity and whose purchase cost is variable based on the user-specified composition of the child entity. The specified template is used similarly to the buy template that appears within tables and choosers. However, the template is centered at the bottom of the form used for editing the child gizmo. This mechanism is needed when you have something like a user-customizable magic item within the d20 System, where the cost is based on the options selected by the user. If empty, no template is associated with the thing. Default: Empty.
xactspecial	(Optional) Integer – When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".
isprivate	(Optional) Boolean – Indicates whether this thing should be kept private from users within the integrated Editor. When a user creates a new thing as a copy of another thing, all existing things derived from the compset are presented for use as the copy source. By making this thing private, it will not appear for selection. Default: "no".

The "thing" element also possesses child elements that define various facets of the thing. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

fieldval	Zero or more "fieldval" elements may appear as defined by the given link. This element specifies the starting value for individual fields within the thing.
arrayval	Zero or more "arrayval" elements may appear as defined by the given link. This element specifies the starting value for individual array and matrix elements within the thing.
usesource	Zero or more "usesource" elements may appear as defined by the given link. This element specifies the sources relied upon for this thing to be accessible to the user.
tag	Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the thing.
bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped when this thing is added to a container.
containerreq	Zero or more "containerreq" elements may appear as defined by the given link. This element specifies any container requirements for the thing.
holdlimit	An optional "holdlimit" element may appear as defined by the given link. This element defines a HoldLimit Tag Expression that restricts the valid set of gear that can be held by the thing.
gear	An optional "gear" element may appear as defined by the given link. This element defines a Gear Script for the thing to calculate its weight.
link	Zero or more "link" elements may appear as defined by the given link. This element specifies the specific pick linkages that exist for this thing.
eval	Zero or more "eval" elements may appear as defined by the given link. This element specifies any Eval Scripts that must be performed for the thing.
evalrule	Zero or more "evalrule" elements may appear as defined by the given link. This element specifies any EvalRule Scripts that must be performed for the thing.
pickreq	Zero or more "pickreq" elements may appear as defined by the given link. This element specifies any dependencies upon other picks within the container.
exprreq	Zero or more "exprreq" elements may appear as defined by the given link. This element specifies any expression-based dependencies upon the state of the container.
prereq	Zero or more "prereq" elements may appear as defined by the given link. This element specifies any pre-requisite tests that are applied to the thing.
child	An optional "child" element may appear as defined by the given link. This element defines the particulars of a child entity that is attached by the thing.
minion	An optional "minion" element may appear as defined by the given link. This element defines the particulars of a minion that is attached by the thing.

The "fieldval" Element

The "fieldval" element defines the starting values to use for various fields within the thing. To initialize elements within arrays and matrices, please see the "arrayval" element (below). The complete list of attributes for this element is below.

field	Id – Unique id of the field for which to specify a starting value.
value	Text – Starting value to assign to the field. If the field is text-based, the value is simply assigned, although it may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.

The "arrayval" Element

The "arrayval" element defines the starting values to use for individual elements within array and matrix fields of the thing. To initialize elements within fields that are not an array or matrix, please see the "fieldval" attribute (above). The complete list of attributes for this element is below.

field	Id – Unique id of the field for which to specify a starting value. If the field is not an array or matrix, an error is reported.
index	Integer – Specifies the row index of the element to be initialized. The first element of arrays and matrices is index zero.
column	(Optional) Integer – Specifies the column of the element to be initialized within a matrix. The first element of matrices is index zero. This attribute is required for matrix elements and may be omitted for array elements. Default: Empty.
value	Text – Starting value to assign to the field element. If the field is text-based, the value is simply assigned, although it may be truncated if it exceeds the defined maximum length for the field. If the field is value-based, the text is converted to a floating point value and assigned.

The "usesource" Element

The "usesource" element specifies a source that this thing is dependent upon. If the designated source is not enabled by the user, this thing will be treated as not existing for the character. You may define a new source on-the-fly via this element by providing a new unique id and the appropriate additional information. However, you should typically avoid doing so, since you cannot control important facets of sources with just-in-time definition. The complete list of attributes for this element is below.

source	Id – Specifies the unique id of the source with which to establish a dependence.
name	(Optional) Text – Name to be displayed to the user for this source. Maximum length is 50 characters. If the source already exists, this attribute can be left empty. Default: Empty.
parent	(Optional) Id – Specifies the unique id of a different source that is treated as the parent of this source. All sources are displayed in a hierarchy within the "Configure Hero" form. If the source already exists, this attribute can be left empty and the behaviors of the existing source are utilized. If empty and the source does not yet exist, the new source is presented to the user as a top-level selection. Default: Empty.

The "holdlimit" Element

The "holdlimit" element defines a HoldLimit Tag Expression for the thing, which restricts the set of things that can be assigned to this thing as held gear. The tag expression is compared against the tags assigned to each prospective piece of gear that the user wants to move. If the tag expression is satisfied, the gear can be held within the thing, else it cannot. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the HoldLimit tag expression.

The "gear" Element

The "gear" element defines a Gear Script for the thing, which is used to calculate the weight of held items and perform additional gear processing on the thing. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Gear script.

The "link" Element

The "link" element defines a linkage to another thing based on the set of possible linkages defined for the underlying components. The complete list of attributes for this element is below.

linkage	Id – Unique id of the linkage that is being setup for this thing.
thing	Id – Unique id of the thing to which the linkage should be established.

The "pickreq" Element

The "pickreq" element defines a dependency on a specific thing whose existence is important within the container. The complete list of attributes for this element is below.

thing	Id – Unique id of the thing upon which the dependency is being established.
iserror	(Optional) Boolean – Indicates whether failure of the requirement should be treated as an error or merely a warning. Default: "yes".
ispreclude	(Optional) Boolean – Indicates whether the specified thing must exist or must not exist within the container in order for the requirement to be satisfied. Preclusion implies that the thing is not allowed to exist. Default: "no".
onlyonce	(Optional) Boolean – Indicates whether the pre-requisite should only be reported to the user a single time if it fails, regardless of the number of times the thing is added to the container. Default: "no".
issilent	(Optional) Boolean – Indicates whether the pre-requisite should report an error message to the user if failed. Default: "no".

The "exprreq" Element

The "exprreq" element defines a dependency on an arithmetic expression that must be satisfied by the container. The complete list of attributes for this element is below.

message	Text – Specifies the message to be reported if the requirement is not satisfied.
iserror	(Optional) Boolean – Indicates whether failure of the requirement should be treated as an error or merely a warning. Default: "yes".
onlyonce	(Optional) Boolean – Indicates whether the pre-requisite should only be reported to the user a single time if it fails, regardless of the number of times the thing is added to the container. Default: "no".
issilent	(Optional) Boolean – Indicates whether the pre-requisite should report an error message to the user if failed. Default: "no".
PCDATA	Text – Specifies an arithmetic expression that is applied to the container to determine whether the requirement is satisfied. The expression must be a valid chunk of script code that can be placed between the parentheses in a standard "if/then" statement (e.g. "if (expr) then").

The "child" Element

The "child" element specifies an entity that is always added as a child gizmo of the thing. The complete list of attributes for this element is below.

entity	Id – Specifies the unique id of the entity to be attached as a child gizmo.

The "child" element also possesses child elements that tailor the entity. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

tag	Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the gizmo when it is created.
bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into the gizmo when it is created.

The "minion" Element

The "minion" element specifies that the thing always attaches a minion. The complete list of attributes for this element is below.

id	Id – Specifies the unique id to be used for the minion attached via this thing.
ownmode	(Optional) Boolean – Indicates whether the minion has an independent creation/advancement mode from its master. In some cases, you will want the minion to possess the same behavior as the master and transition with the master. However, if the minion is constructed as an independent character, you may want to handle advancement separately. Default: "yes".
isinherit	(Optional) Boolean – Indicates whether the minion inherits the set of enabled sources from its master. If inheritance is active, the source selections for the master are locked in for the minion, with any changes to the master being immediately reflected within the minion. Default: "no".

The "minion" element also possesses child elements that tailor the minion. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

tag	Zero or more "tag" elements may appear as defined by the given link. This element specifies any tags that are automatically assigned to the minion when it is created.
bootstrap	Zero or more "bootstrap" elements may appear as defined by the given link. This element specifies any things that are automatically bootstrapped into the minion when it is created.

Example

The following example demonstrates what a "thing" element might look like. All default values are assumed for optional attributes.

```
<thing id="MyThing" name="Sample" compset="Race" isunique="yes"
  description="Description goes here">
  <fieldval field="FieldId" value="42"/>
  <tag group="Helper" tag="MyTag"/>
  <bootstrap thing="MyAbility"/>
  <eval phase="Setup" priority="5000">
    ~script code goes here
    </eval>
  </thing>
```

Category: Kit Reference

Portal Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Multiple Sources

Contents

The "portal" Element

Within the hierarchy of visual elements, the individual elements that the user interacts with are portals. There is a wide assortment of different portal types that can be employed for different purposes. Each separate portal is specified through the use of a "portal" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the portal. This id is used in all references to the portal.
style	Id – Specifies the unique id of the style to utilize with this portal. The style must be compatible with the portal type (e.g. a label style must be used with a label portal).
tiptext	(Optional) Text – Description information to be shown to the user when the user pauses the mouse over the portal. If empty, nothing is shown. Default: Empty.
isheader	(Optional) Boolean – Indicates whether this portal is utilized within a dual-purpose header as part of a table. Default: "no". **NOTE!** This attribute is only permitted on portals within templates that serves as dual-purpose headers. **NOTE!** Field-based portals may **never** be designated for use within a header, as there is no pick/thing associated with dual-purpose templates.
showinvalid	(Optional) Boolean – Indicates whether the text used within the portal should be automatically changed to the built-in "lblwarning" color when the portal references an invalid pick. This behavior only applies to suitable portal types, including labels, incrementers, checkboxes, and menus. Default: "no".
showdisabled	(Optional) Boolean – Indicates whether the text used within the portal should be automatically changed to the built-in "lbldisable" color when the portal references a pick whose pre-requisites are not satisfied. This behavior only applies to suitable portal types, including labels, incrementers, checkboxes, and menus. Default: "yes".
width	(Optional) Integer – Specifies the default width to use for the portal. In general, portal dimensions should only be controlled via the Position script of the containing visual element. If zero, the default auto-sizing behaviors are employed. Default: "0".
height	(Optional) Integer – Specifies the default height to use for the portal. In general, portal dimensions should only be controlled via the Position script of the containing visual element. If zero, the default auto-sizing behaviors are employed. Default: "0".

The "portal" element also possesses child elements that define the specifics of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! With the exception of the "live" and "mouseinfo" elements, exactly **one** of these child elements may be specified for each portal. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of portal that is being defined and its characteristics. You may include up to one "live" and/or "mouseinfo" elements after the single child element that specifies the portal.

label	An optional "label" element may appear as defined by the given link. This element specifies the details of a label portal.
image_field	An optional "image_field" element may appear as defined by the given link. This element specifies the details of a field-based image portal.
image_user	An optional "image_user" element may appear as defined by the given link. This element specifies the details of an image portal containing user-selected images.
image_literal	An optional "image_literal" element may appear as defined by the given link. This element specifies the details of an image portal containing a static image.
image_reference	An optional "image_reference" element may appear as defined by the given link. This element specifies the details of an image portal that references a field-based image.

incrementer	An optional "incrementer" element may appear as defined by the given link. This element specifies the details of an incrementer portal.
edit	An optional "edit" element may appear as defined by the given link. This element specifies the details of an edit portal.
edit_date	An optional "edit_date" element may appear as defined by the given link. This element specifies the details of an editdate portal.
checkbox	An optional "checkbox" element may appear as defined by the given link. This element specifies the details of a checkbox portal.
menu_literal	An optional "menu_literal" element may appear as defined by the given link. This element specifies the details of a menu portal consisting of a fixed set of options.
menu_array	An optional "menu_array" element may appear as defined by the given link. This element specifies the details of an array-based menu portal.
menu_things	An optional "menu_things" element may appear as defined by the given link. This element specifies the details of a thing-based menu portal.
action	An optional "action" element may appear as defined by the given link. This element specifies the details of an action portal.
region	An optional "region" element may appear as defined by the given link. This element specifies the details of a region portal.
separator	An optional "separator" element may appear as defined by the given link. This element specifies the details of a separator portal.
chooser_table	An optional "chooser_table" element may appear as defined by the given link. This element specifies the details of a table-based chooser portal.
table_fixed	An optional "table_fixed" element may appear as defined by the given link. This element specifies the details of a non-editable table portal.
table_dynamic	An optional "table_dynamic" element may appear as defined by the given link. This element specifies the details of a table portal to which the user can add arbitrary items.
table_auto	An optional "table_auto" element may appear as defined by the given link. This element specifies the details of a table portal to which a specific item can be added.
setting_edit	An optional "setting_edit" element may appear as defined by the given link. This element specifies the details of a special portal for editing configuration settings.
setting_summary	An optional "setting_summary" element may appear as defined by the given link. This element specifies the details of a special portal for showing a summary of configuration settings.
alliance	An optional "alliance" element may appear as defined by the given link. This element specifies the details of a special portal for controlling the alliance state of an actor.
output_label	An optional "output_label" element may appear as defined by the given link. This element specifies the details of a label portal for sheet output.
output_image	An optional "output_image" element may appear as defined by the given link. This element specifies the details of an image portal for sheet output.
output_table	An optional "output_table" element may appear as defined by the given link. This element specifies the details of a table portal for sheet output.
output_dots	An optional "output_dots" element may appear as defined by the given link. This element specifies the details of a special label portal for sheet output.
output_separator	An optional "output_separator" element may appear as defined by the given link. This element specifies the details of a separator portal for sheet output.
live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression for the portal.
mouseinfo	An optional "mouseinfo" element may appear as defined by the given link. This element defines a MouseInfo Script for the portal.

The "live" Element

The "live" element defines a Live Tag Expression for the portal that determines whether the portal is applicable based on the prevailing conditions. In general, portals should be controlled via scripts using the "visible" target reference instead of using this mechanism. The complete list of attributes for this element is below.

PCDATA TagExpr – Specifies the code comprising the Live tag expression.

The "mouseinfo" Element

The "mouseinfo" element defines a MouseInfo Script for the portal that synthesizes text for display to the user whenever the user pauses the mouse over the portal. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the MouseInfo script.

Example

The following example demonstrates what various "portal" elements might look like. All default values are assumed for optional attributes.

```
<portal id="hair" style="editNormal">
  <edit field="perHair"/>
  </portal>

<portal id="name" style="chkNormal" showinvalid="yes"
      tiptext="Click to equip this item">
  <checkbox field="grIsEquip" dynamicfield="grStkName"/>
  </portal>

<portal id="menu" style="menuNormal">
  <menu_things field="adjChosen" component="none" maxvisible="20"
        usepicksfield="adjUsePick" candidatefield="adjCandid">
    <candidate></candidate>
    </menu_things>
  </portal>

<portal id="gender" style="menuNormal">
  <menu_literal field="perGender">
    <choice value="0" display="Gender: Male"/>
    <choice value="1" display="Gender: Female"/>
    </menu_literal>
  </portal>

<portal id="textlist" style="menuNormal">
  <menu_array field="conTextSel" array="conList" maxvisible="10"/>
  </portal>

<portal id="stRace" style="chsNormal" width="110">
  <chooser_table component="Race" choosetemplate="LargeItem">
    <chosen>
      @text = "Race: " & field[name].text
      </chosen>
    <titlebar>
      @text = "Choose the race for your character"
      </titlebar>
    </chooser_table>
  </portal>
```

Category: Kit Reference

Label Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

The "label" Element

When you want to display text to the user, it's usually easiest to utilize a label portal, which is specified through the use of a "label" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the label contents may be employed within a given label portal. That means you may use **either** the "text" attribute, the "field" attribute, **OR** the "labeltext" script to define the contents. Use of multiple mechanisms will result in a compilation error.

text	(Optional) Text – Specifies a string of literal text to be displayed within the label. Default: Empty.
field	(Optional) Id – Specifies the unique id of the field whose value is to be displayed within the label. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based label is not allowed. Default: Empty.
ismultiline	(Optional) Boolean – Indicates whether the label text is to be treated as multi-line or merely a single line of output. Default: "no".
scrollable	(Optional) Boolean – Indicates whether the label text is large enough to require a vertical scroller be included that allows the user to scroll through the contents. Default: "no".
istitle	(Optional) Boolean – Indicates whether the label text is being used as a title, which entails special automatic sizing behaviors for proper handling. Default: "no".

The "label" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

labeltext	An optional "labeltext" element may appear as defined by the given link. This element defines a Label Script that is used for synthesizing the text to be output.

The "labeltext" Element

The "labeltext" element defines a Label Script for the portal. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Label script.

Example

The following example demonstrates what a label portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="lblNormal" showinvalid="yes">
  <label field="name"/>
  </portal>

<portal id="cost" style="lblNormal">
  <label>
    <labeltext>
      @text = field[grCost].text
      </labeltext>
    </label>
  </portal>
```

Category: Kit Reference

ImageField Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "image_field" Element

The role of the "image_field" element is to display an image to the user, where the actual image is determined by the contents of a field. If you have a situation where you need to display a different image based on the situation, you have two choices. The first option is to have separate image portals for each and manage the visibility so only the correct one is shown. The second option is to have one field-based image portal and select the proper image via a script. The latter approach is often easier. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field must be a text-based field and its contents are assumed to specify the filename of a bitmap image within the data file folder for the game system. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed.
istransparent	(Optional) Boolean – Indicates whether the image should be treated as transparent, wherein the pixel color at position 0,0 is considered transparent throughout the image. Default: "no".

Example

The following example demonstrates what a field-based image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_field field="imagefile"/>
  </portal>
```

Category: Kit Reference

ImageUser Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "image_user" Element

The role of the "image_user" element is to allow the user to select the image to be displayed and show it within the portal. Until the user selects an image, a placeholder image is shown instead. This portal type is intended for use in allowing the user to select a character portrait and related situations. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents manage the image to to be displayed within the portal. The field must be a value-based field of "user" style, and its contents should never be modified by script. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed. **NOTE!** User-selected images are tracked internally with the portfolio, so no link to the original image file exists.

Example

The following example demonstrates what a user image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_user field="userimage"/>
  </portal>
```

Category: Kit Reference

ImageLiteral Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "image_literal" Element

The role of the "image_literal" element is to display a fixed image to the user, where the image is dictated by the portal definition. Whenever you need to show an icon to the user, such as when indicating a particular condition is present, you can use a literal image portal. The complete list of attributes for this element is below.

image	Text – Specifies the filename of a bitmap image within the data file folder for the game system. The given image is displayed within the portal.
istransparent	(Optional) Boolean – Indicates whether the image should be treated as transparent, wherein the pixel color at position 0,0 is considered transparent throughout the image. Default: "no".
isbuiltin	(Optional) Boolean – Indicates whether the image file is a "built-in" file provided by HL for easy re-use. Default: "no".

Example

The following example demonstrates what a literal image portal might look like. All default values are assumed for optional attributes.

```
<portal id="heldby" style="imgNormal">
  <image_literal image="gearinfo.bmp" istransparent="yes"/>
  <mouseinfo mousepos="middle+above">
    call InfoHeld
    </mouseinfo>
  </portal>
```

Category: Kit Reference

ImageReference Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "image_reference" Element

It is not possible to copy a field-based image (including user-added images) to another field. However, there are times when you'll want to do exactly that. Consequently, the Kit provides a mechanism for establishing a reference to an image field. The role of the "image_reference" element is to display the contents of one of these field references. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field must be a value-based field and it must contain a reference to another field that identifies an image. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a reference-based image is not allowed.

Example

The following example demonstrates what a referenace-based image portal might look like. All default values are assumed for optional attributes.

```
<portal id="image" style="imgNormal">
  <image_reference field="imageref"/>
  </portal>
```

Category: Kit Reference

Incrementer Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

The "incrementer" Element

The "incrementer" element comes into play when you want to allow the user to modify a value via "plus" and "minus" adjustments. This approach is ideal for managing attributes, skill ratings, consumption of resources, and a variety of other situations. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents dictate the value displayed within the portal. The field must be a value-based field and the field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an incrementer is not allowed.
interval	(Optional) Integer – Specifies the adjustment to be applied whenever the user clicks on the '+' and '-' buttons for the incrementer. Default: "1".

Example

The following example demonstrates what an incrementer portal might look like. All default values are assumed for optional attributes.

```
<portal id="adjust" style="incrSimple">
  <incrementer field="adjUser"/>
  </portal>
```

Category: Kit Reference

Edit Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "edit" Element

The "edit" element is used to allow the user to directly enter a value or text that is then saved into a field. This portal is ideal for allowing the user to enter names or descriptions, as well as entering a wide-ranging or arbitrary value. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field where the contents to be edited are both retrieved and stored. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an edit portal is not allowed.
maxlength	(Optional) Integer – Specifies the maximum number of characters that can be entered for this field. If zero, a numeric value will be edited. Default: "0".
ismultiline	(Optional) Boolean – Indicates whether the field should be edited as if it contains multi-line text. This attribute only applies to text-based edit portals. Default: "no".
readonly	(Optional) Boolean – Indicates whether the information within the portal can be edited by the user or is displayed as read-only. Default: "no".
format	(Optional) Set – Designates any special numeric formatting to be enforced for a value-based field. Must be one of these values: ▪ integer – The contents are edited as an integer value. ▪ float – The contents are edited as a floating-point value. ▪ any – No restrictions are imposed on the contents. ▪ Default: "any".
signed	(Optional) Boolean – Indicates whether the user can specify a negative value when a value-based field is being edited. Default: "no".

Example

The following example demonstrates what an edit portal might look like. All default values are assumed for optional attributes.

```
<portal id="hair" style="editNormal">
  <edit field="perHair"/>
  </portal>
```

Category: Kit Reference

EditDate Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "edit_date" Element

The "edit_date" element allows the user to edit a structured date or time that adheres to the syntactic rules defined for the game system. If a "game world" date or time is edited, then the proper pieces are presented to the user, as dictated by the structure specified for each within the definition file. If a "real world" date or time is edited, then the traditional pieces are presented. This makes it possible to ensure the user enters a syntactically valid date/time, although semantic rules are not currently enforced. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field where the date/time value to be edited is both retrieved and stored. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, an editdate portal is not allowed.
readonly	(Optional) Boolean – Indicates whether the information within the portal can be edited by the user or is displayed as read-only. Default: "no".
format	(Optional) Set – Designates the formatting behavior to be utilized. Must be one of these values: ▪ gamedate – The contents are edited as a game system date. ▪ gametime – The contents are edited as a game system time. ▪ realdate – The contents are edited as a real world date. ▪ realtime – The contents are edited as a real world time. ▪ Default: "realdate".

Example

The following example demonstrates what an editdate portal might look like. All default values are assumed for optional attributes.

```
<portal id="date" style="editDate">
  <edit_date field="thedate" format="gamedate"/>
  </portal>
```

Category: Kit Reference

Checkbox Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

The "checkbox" Element

The "checkbox" element is useful whenever the user has a choice between two clearly opposite states, such as on/off, enable/disable, show/hide, etc. In addition to the traditional visual presentation of text with a box next to it, the Kit allows you to use checkboxes to present to alternate visual states. For example, within the World of Darkness data files, the abilities to promote items to the top of a list and toggle inclusion within printouts are checkboxes that merely toggle between two states and change the visuals accordingly. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents dictate whether the checkbox is in the on or off state. The field must be a value-based field, with a non-zero value indicating "on" and a zero value indicating "off". The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a checkbox is not allowed.
message	(Optional) Text – Specifies the message text to display next to the actual box. If empty, no text is displayed. Default: Empty.
dynamicfield	(Optional) Id – Specifies the unique id of a different field from which the message text will be pulled. This allows the message text to be dynamically determined via scripts. If empty, no dynamic field is specified. Default: Empty.
readonly	(Optional) Boolean – Indicates whether the checkbox portal is unable to be changed by the user. Default: "no".

Example

The following example demonstrates what a checkbox portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="chkNormal" showinvalid="yes"
      tiptext="Click to equip this weapon">
  <checkbox field="grIsEquip" dynamicfield="grStkName"/>
  </portal>
```

Category: Kit Reference

MenuLiteral Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "menu_literal" Element

Menus are useful whenever you need the user to select exactly one item from a collection of options. The role of the "menu_literal" element is to allow you to specify a fixed set of options to choose from. An classic example is the selection of gender on the "Personal" tab, where a menu allows the user to select either male or female. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a text-based field and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
maxvisible	(Optional) Integer – Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".
allowuservalue	(Optional) Boolean – Indicates whether the user is allowed to type in a custom value for use within the menu. If the user does this, the value saved for the menu is simply the text entered by the user. Default: "no".

The "menu_literal" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

choice	One or more "choice" elements must appear as defined by the given link. This element defines the individual options that the user can select from.

The "choice" Element

The "choice" element defines an option that is listed within the menu for the user to select. Each choice has two facets: the value shown to the user and the value used internally. The sequence in which these elements is defined dictates the sequence in which they will be listed for the user. The complete list of attributes for this element is below.

display	Text – Specifies the text to be displayed for this choice.
value	(Optional) Text – Specifies the text to be tracked internally for this choice. When the user selects the choice, the value is saved into the associated field. If empty, the "display" text is used as the value. If the field used by the menu is value-based, all "value" attributes must be numeric values. Default: Empty.

Example

The following example demonstrates what a literal menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="gender" style="menuNormal">
  <menu_literal field="perGender">
    <choice value="0" display="Gender: Male"/>
    <choice value="1" display="Gender: Female"/>
    </menu_literal>
  </portal>
```

Category: Kit Reference

MenuArray Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "menu_array" Element

The "menu_array" element allows you to create a menu whose options are dynamically determined via scripts. These scripts setup the contents of the array that dictates the available options. The complete list of attributes for this element is below.

field	Id – Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a text-based field and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
array	Boolean – Specifies the unique id of the array-based field to be used to drive the available options. The field must be text-based, must be an array, and must exist within the pick/thing associated with the containing template.
maxvisible	(Optional) Integer – Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".

Example

The following example demonstrates what an array-based menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="menutext" style="Menu" width="150">
  <menu_array field="pwmTextSel" array="pwmList" maxvisible="9"/>
  </portal>
```

Category: Kit Reference

MenuThings Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

Contents

The "menu_things" Element

The "menu_things" element is used when the user needs to select an item that exists within the data files. For example, an in-play adjustment can select an attribute or skill that the actor possesses, or an ability can select a weapon type that receives a bonus. The common thread is that the options presented in the menu are either things or picks within the data files. The complete list of attributes for this element is below.

IMPORTANT! Within a thing-based menu, the selected thing is **not** added to the container as a new pick. The thing is merely identified for reference and can be accessed via the menu, but no new pick appears within the container. Only tables and choosers can actually add new picks to a container.

field	Id – Specifies the unique id of the field whose contents reflect the current selection from the menu. The field must be a value-based field of style "menu" and must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a menu is not allowed.
component	Id – Specifies the unique id of the component that all choices must be derived from.
defthing	(Optional) Id – Specifies the unique id of the thing to pre-select as the default choice within the menu. If empty, no default selection is made. Default: Empty.
usepicks	(Optional) Set – Designates whether the menu is populated with things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values: ▪ thing – The menu is populated with things. ▪ container – The menu is populated with picks from the container of the pick associated with the template. ▪ hero – The menu is populated with picks from the actor. ▪ actor – Same as "hero". ▪ Default: "thing".
sortset	(Optional) Id – Specifies the unique id of the sort set to be used to determine the sequence in which the items are listed in the menu. If empty, the items are listed alphabetically. Default: Empty.
maxvisible	(Optional) Integer – Specifies the maximum number of items that will be visible at one time within the menu when the user opens it for selection. If there are more items to choose from, a scroller will allow the user to access them. Default: "5".
usepicksfield	(Optional) Id – Specifies the unique id of a value-based field that dynamically dictates when the menu is populated with things or picks. If empty, the "usepicks" attribute dictates the behavior. If specified, the field value dictates the behavior according to the list below. Default: Empty. ▪ 0 – The menu is populated with things. ▪ 1 – The menu is populated with picks from the container of the pick associated with the template. ▪ 2 – The menu is populated with picks from the actor.
candidatefield	(Optional) Id – Specifies the unique id of a text-based field that contains the Candidate tag expression to be used when populating the menu options. Default: Empty. IMPORTANT! This mechanism is secondary to the "candidate" element, so the field is **ignored** if the "candidate" element is non-empty.
namefield	(Optional) Id - Specifies the unique id of a text-based field that is used instead of the "name" of each thing shown within the array. This makes it possible to customize the name to be displayed in the menu differently from the standard name shown for the thing.

The "menu_things" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

candidate	An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
change	An optional "change" element may appear as defined by the given link. This element defines a Change Script for the portal.

The "candidate" Element

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Candidate tag expression.

The "change" Element

The "change" element defines a Change Script for the portal that is invoked whenever the user selects a new choice from the list of options. This script allows the implications of the new selection to be integrated and the display updated. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Change script.

Example

The following example demonstrates what a thing-based menu portal might look like. All default values are assumed for optional attributes.

```
<portal id="menu" style="menuNormal">
  <menu_things field="adjChosen" component="none" maxvisible="20"
        usepicksfield="adjUsePick" candidatefield="adjCandid">
    <candidate></candidate>
    </menu_things>
  </portal>
```

Category: Kit Reference

Action Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "action" Element

Action portals behave like buttons and always trigger some sort of behavior. That behavior could be built into HL (e.g. delete a pick) or completely defined by you, but some sort of action is invoked. Action portals are specified via the "action" element. The complete list of attributes for this element is below.

IMPORTANT! Many action portals have behaviors that rely on an associated pick. These portals assume they are defined within the context of a template, and the associated pick is dictated by that template.

action	(Optional) Set – Designates the behavior to be invoked when the action portal is triggered by the user. Must be one of these values: ▪ delete – Deletes the associated pick. Only useful within tables for allowing the user to delete a pick that they added. ▪ info – Displays detailed information about the associated pick via the associated MouseInfo script. If no script is defined, default behavior shows the name, failed pre-requisites, and description for the associated pick. ▪ edit – Brings up a form within which the user can edit the contents of a gizmo, where the gizmo is the child of the associated pick. ▪ form – Brings up the modal form specified within the "form" attribute. ▪ trigger – Invokes the Trigger script specified via the "trigger" child element. ▪ gear – Displays a menu allowing the user to move gear between holders. ▪ notes – Brings up a form within which the user can edit the contents of the field designated by the "notes" attribute. ▪ load – Loads the actor containing the associated pick for manipulation by the user. ▪ lock – Transitions the actor containing the associated pick into advancement mode. ▪ unlock – Transitions the actor containing the associated pick into creation mode. ▪ master – Loads the master of the actor containing the associated pick, making it the active actor. ▪ minion – Loads the minion attached by the associated pick, making it the active actor. If the "minion" attribute specifies a unique id, the designated minion is loaded instead. ▪ getgear – Displays a menu that allows the user to move gear between actors. ▪ combatstart – Triggers the start of a new combat within the Tactical Console. ▪ combatend – Triggers the end of an existing combat within the Tactical Console. ▪ newturn – Transitions to a new combat turn within the Tactical Console. ▪ initchange – Incorporates user-made changes to the initiatives of actors within the Tactical Console. ▪ integrate – Integrates all pending actors into an existing combat within the Tactical Console. ▪ dashsort – Triggers a re-sort of all actors shown within the Dashboard. ▪ Default: "delete".
buttontext	(Optional) Text – Specifies the text to draw over whatever bitmap is used for the button. Although buttons with text are often larger that other buttons, they are also much easier to create. If empty, no text is drawn over the bitmap. Default: Empty.
confirm	(Optional) Text – Specifies the text to be displayed as a confirmation message before the triggered behavior is actually performed. This attribute is only utilized for the "trigger", "delete", "edit", and "form" action types. If empty, the triggered behavior is performed immediately - without any confirmation check. Default: Empty.
form	(Optional) Id – Specifies the unique id of the form to be displayed when the portal is triggered. This attribute only applies to the "form" action type and is required for that action type. Default: Empty.
field	(Optional) Id – Specifies the unique id of the field that contains the notes text to be edited. This attribute only applies to the "notes" action type and is required for that action type. Default: Empty.
minion	(Optional) Id – Specifies the unique id of the minion to be switched to when the "minion" action portal is triggered. This attribute only applies to the "minion" action type, but it is **not** required for that action type. If specified, the indicated minion is accessed. If not specified, the minion attached directly by the pick associated

with the portal (via the containing template) is accessed. Default: Empty.

The "action" element also possesses child elements that define additional facets of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

trigger	An optional "trigger" element may appear as defined by the given link. This element defines a Trigger Script for the portal.

The "trigger" Element

The "trigger" element defines a Trigger Script for the portal that is invoked whenever the user triggers the portal. This script only applies to the "trigger" action type and allows adjustments to be applied to usage pools, such as those for tracking damage, managing journal entries, etc. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Trigger script.

Example

The following example demonstrates what an action portal might look like. All default values are assumed for optional attributes.

```
<portal id="delete" style="actDelete"
      tiptext="Click to delete this equipment">
  <action action="delete"/>
  </portal>

<portal id="addxp" style="actSmall">
  <action action="trigger" buttontext="Add XP">
    <trigger>
      ~add the XP to both the journal entry's and hero's usage pools
      perform usagepool[JrnlXP].adjust[field[jrnXP].value]
      perform hero.usagepool[TotalXP].adjust[field[jrnXP].value]
      perform field[jrnXP].reset
      </trigger>
    </action>
  </portal>
```

Category: Kit Reference

Region Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "region" Element

The role of the "region" element is to allow you to designate a rectangular region that can have a suitable border drawn around it. When you want to put a border around a collection of portals to visually group them, you can use a region element. There are no attributes or child elements for this element.

Example

The following example demonstrates what a region portal might look like. All default values are assumed for optional attributes.

```
<portal id="border" style="rgnBorder">
  <region/>
  </portal>
```

Category: Kit Reference

Separator Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

The "separator" Element

The role of the "separator" element is to insert a vertical or horizontal bar between groupings of portals, acting as a visual separator between them. The complete list of attributes for this element is below.

isvertical	(Optional) Boolean – Indicates whether the separator should be drawn horizontally or vertically. Default: "no".

Example

The following example demonstrates what a separator portal might look like. All default values are assumed for optional attributes.

```
<portal id="separator" style="sepHorz">
  <separator isvertical="no"/>
  </portal>
```

Category: Kit Reference

ChooserTable Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

Contents

The "chooser_table" Element

Choosers are similar to thing-based menus in some ways, as they allow the user to select one thing or pick from a list that is determined dynamically. One key difference with choosers is that any selected thing/pick is added to the container as a new pick. If a pick is selected, a new pick derived from the same thing is added. Another key difference is that the available things/picks are displayed for selection in a "choose form", allowing each object to be presented with detailed information. The "chooser" mechanism is ideal for selecting facets like race, profession, archetype, etc. Each chooser is defined via the use of the "chooser_table" element. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component that all selectable objects must be derived from.
choosetemplate	Id – Specifies the unique id of the template to be used for displaying selectable objects.
choosepicks	(Optional) Set – Designates whether the selectable objects consist of things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values: ▪ thing – The selectable objects are things. ▪ container – The selectable objects are picks from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used. ▪ hero – The selectable objects are picks from the active actor. ▪ actor – Same as "hero". ▪ Default: "thing".
choosesortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing all of the objects presented for selection. If empty, all objects are sorted by name. Default: Empty.
choosegapx	(Optional) Integer – Specifies the gap along the X-axis to insert between items presented for selection. Default: "0".
choosegapx	(Optional) Integer – Specifies the gap along the Y-axis to insert between items presented for selection. Default: "0".
descwidth	(Optional) Integer – Specifies the width of the reserved "description" area on the right within the choose form. Some items need more width for lengthy descriptions and some do not, so you can control this as you see fit. Default: "250".
buytemplate	(Optional) Id – Specifies the unique id of the template to be shown in the lower right corner of the choose form for controlling the details of a purchase transaction. If empty, no buy template is utilized. Default: Empty.
xactspecial	(Optional) Integer – When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".

linkage	(Optional) Id – Specifies the unique id of a thing that will be used as a linkage. When a new pick is added via the chooser, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
showupdate	(Optional) Boolean – Indicates whether the chooser needs to be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user, such as through color highlighting. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.
candidatepick	(Optional) Id – Specifies the unique id of a pick that will contain a dynamically generated Candidate tag expression for use in determining the list of available objects to choose from. If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
candidatefield	(Optional) Id – Specifies the unique id of a text-based field that contains the Candidate tag expression used to determine the list of available objects to choose from. This field must exist within the pick identified by the "candidatepick" attribute (above). If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
prereqtarget	(Optional) Set – Designates the container against which all pre-requisite tests need to be performed when determining the list of items available for selection. When displacement is utilized, pre-requisites need to be tested against the container to which the new picks will ultimately be added. Must be one of these values: ▪ container – The default parent container is used. ▪ parent – The next parent up the hierarchy is used, which parallels the corresponding displacement target. ▪ hero – The top-level hero is used, which parallels the corresponding displacement target. ▪ Default: "container".
empty	(Optional) Text – Specifies the text message to be displayed if the user attempts to select an option and there are no available items to choose from. If empty, a default message is displayed. Default: Empty.

NOTE! Choosers possess a "buy" template, but there is no way to properly "sell" an item selected via a chooser. The reason for this is to allow the "buy" template to be used for customization purposes instead of actually buying and selling gear. The same mechanism can be used to allow the user to configure the item selected via the chooser, such as providing an edit or menu portal to specify an important facet of the selected item.

The "chooser_table" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

candidate	An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
needtag	Zero or more "needtag" elements may appear as defined by the given link. This element defines a tag relationship that must exist between a prospective object and the container in order to list the object among the available items.
denytag	Zero or more "denytag" elements may appear as defined by the given link. This element defines a tag relationship that must **not** exist between a prospective object and the container in order to list the object among the available items.
xacttag	Zero or more "xacttag" elements may appear as defined by the given link. This element defines a tag that is assigned to the transaction pick while the choose form is visible.
secondary	An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.
existence	An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.
autotag	Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to each added thing.
chosen	An optional "chosen" element may appear as defined by the given link. This element defines a Chosen Script for the portal.
titlebar	An optional "titlebar" element may appear as defined by the given link. This element defines a TitleBar Script for the portal.

description	An optional "description" element may appear as defined by the given link. This element defines a Description Script for the portal.
change	An optional "change" element may appear as defined by the given link. This element defines a Change Script for the portal.

The "candidate" Element

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Candidate tag expression.

The "needtag" Element

The "needtag" element defines a tag relationship that must exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it has at least one matching tag with the same id in a separate tag group. If the tag is not found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

container	Id – Specifies the unique id of the tag group to utilize within the container.
thing	Id – Specifies the unique id of the tag group to check within the thing/pick.
usehero	(Optional) Boolean – Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

The "denytag" Element

The "denytag" element defines a tag relationship that must **not** exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it does not possess any matching tags with the same ids in a separate tag group. If any matching tags are found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

container	Id – Specifies the unique id of the tag group to utilize within the container.
thing	Id – Specifies the unique id of the tag group to check within the thing/pick.
usehero	(Optional) Boolean – Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

The "xacttag" Element

The "xacttag" element specifies a tag that is automatically added to the transaction pick while the choose form is shown. These tags allow you to indicate contextual information about where the buy template is being used so that you can tailor the behavior appropriately. The complete list of attributes for this element is below.

tag	Id – Specifies the unique id of the tag to define within the tag group "transact".

The "secondary" Element

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "existence" Element

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "chosen" Element

The "chosen" element defines a Chosen Script for the portal, which synthesizes the text to be displayed as the chosen item within the portal. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Chosen script.

The "titlebar" Element

The "titlebar" element defines a TitleBar Script for the portal, which synthesizes the text to be displayed at the top of the choose form. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the TitleBar script.

The "description" Element

The "description" element defines a Description Script for the portal, which synthesizes the text to be displayed within the description region of the choose form for the currently selected item on the left. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Description script.

The "change" Element

The "change" element defines a Change Script for the portal that is invoked whenever the user selects a new choice from the list of options. This script allows the implications of the new selection to be integrated and the display updated. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Change script.

Example

The following example demonstrates what a choosertable portal might look like. All default values are assumed for optional attributes.

```
<portal id="stRace" style="chsNormal" width="110">
  <chooser_table component="Race" choosetemplate="LargeItem">
    <chosen><![CDATA[
      if (@ispick = 0) then
        @text = "{text ff0000}Select Race"
      else
        @text = "Race: " & field[name].text
        endif
      ]]></chosen>
    <titlebar>
      @text = "Choose the race for your character"
      </titlebar>
    </chooser_table>
  </portal>
```

Category: Kit Reference

TableFixed Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

Contents

The "table_fixed" Element

Fixed tables present a list of items to the user that cannot be modified by the user through the table. This is ideal for displaying a list of character attributes or a summary of the special abilities possessed by a character. Since they cannot be modified, fixed tables have only a set of behaviors for showing the selected items to the user. Each fixed table is defined via the use of the "table_fixed" element. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component that all shown objects must be derived from.
showtemplate	Id – Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
showpicks	(Optional) Set – Designates the source from which the picks shown are retrieved from. Must be one of these values: ▪ container – The picks shown are from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used. ▪ hero – The picks shown are from the active actor. ▪ actor – The picks shown represent all actors in the entire portfolio. ▪ lead – The picks shown represent all lead actors in the entire portfolio. ▪ minion – The picks shown are all immediate minions for the active actor. ▪ Default: "container".
showsortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
showgapx	(Optional) Integer – Specifies the gap along the X-axis to insert between items that exist within the table. Default: "0".
showgapy	(Optional) Integer – Specifies the gap along the Y-axis to insert between items that exist within the table. Default: "0".
columns	(Optional) Integer – Specifies the number of columns of data to display within the table. Default: "1".
scrollable	(Optional) Boolean – Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
headertemplate	(Optional) Id – Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
headerpick	(Optional) Id – Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
footertemplate	(Optional) Id – Specifies the unique id of the template to be used for a footer at the bottom of the table. This allows you to put information at the bottom of the table that is always tied to the table for sizing and positioning purposes. If empty, no footer is displayed for the table. Default: Empty.
showfixedlast	(Optional) Boolean – Indicates whether all non-deletable picks within the table are sorted to the end of the list of picks shown. Default: "no".
	(Optional) Boolean – Indicates whether the items in the table can be re-ordered by the user. If enabled, the specified component must designate a suitable ordering field or a separate component with such a field

allowuserorder	must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
ordercomponent	(Optional) Id – Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
alwaysupdate	(Optional) Boolean – Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_fixed" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

list	An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
headertitle	An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.

The "list" Element

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the List tag expression.

The "headertitle" Element

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the HeaderTitle script.

Example

The following example demonstrates what a fixed table portal might look like. All default values are assumed for optional attributes.

```
<portal id="baAttrib" style="tblInvis">
  <table_fixed component="Attribute" scrollable="no"
      showtemplate="baAttrPick" showsortset="explicit">
    <headertitle>
      @text = "Attributes"
      </headertitle>
    </table_fixed>
  </portal>
```

Category: Kit Reference

TableDynamic Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

Contents

The "table_dynamic" Element

Dynamic tables allow the user to select the items to be added to the table. As such, they effectively have one set of behaviors for showing the selected items and a separate set of behaviors for selecting the items. A choose form is used to present the list of available items for selection, which allows for detailed information to be shown for each item. When things are added to a table, a new pick is added to the container that is derived from the selected thing. Each dynamic table is defined via the use of the "table_dynamic" element. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component that all objects must be derived from, both selectable and shown within the table.
showtemplate	Id – Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
showsortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
showgapx	(Optional) Integer – Specifies the gap along the X-axis to insert between items that exist within the table. Default: "0".
showgapy	(Optional) Integer – Specifies the gap along the Y-axis to insert between items that exist within the table. Default: "0".
choosetemplate	Id – Specifies the unique id of the template to be used for displaying available objects that the user can choose from.
choosepicks	(Optional) Set – Designates whether the selectable objects consist of things or picks, and, if the latter, where the list of picks is retrieved from. Must be one of these values: ▪ thing – The selectable objects are things. ▪ container – The selectable objects are picks from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used. ▪ hero – The selectable objects are picks from the active actor. ▪ Default: "thing".
choosesortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing all of the objects presented for selection. If empty, all objects are sorted by name. Default: Empty.
choosegapx	(Optional) Integer – Specifies the gap along the X-axis to insert between items presented for selection. Default: "0".

choosegapy	(Optional) Integer – Specifies the gap along the Y-axis to insert between items presented for selection. Default: "0".
descwidth	(Optional) Integer – Specifies the width of the reserved "description" area on the right within the choose form. Some items need more width for lengthy descriptions and some do not, so you can control this as you see fit. Default: "250".
columns	(Optional) Integer – Specifies the number of columns of data to display within the table. Default: "1".
scrollable	(Optional) Boolean – Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
ismultiadd	(Optional) Boolean – Indicates whether the user can add multiple items at a time to the table, without leaving the choose form. Default: "yes".
allowstack	(Optional) Boolean – Indicates whether the user is allowed to stack items within the table, subject to the restrictions imposed for each item. Default: "yes".
addtemplate	(Optional) Id – Specifies the unique id of the template to be used for the "add" item that always appears at the bottom of a dynamic table and that users will click on to add an item to the table. This allows detailed controlled when the simple "additem" script is not sufficient. If empty, the "additem" element must be specified. Default: Empty.
addpick	(Optional) Id – Specifies the unique id of a thing that is associated with the "add" item at the bottom of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "addtemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
addspace	(Optional) Integer – Specifies the additional vertical space to be inserted when displaying the simple "add" item at the bottom of the table by using the "additem" script. The height of the item is based on the font height of the text shown, so this attribute allows you to insert additional padding if you wish. Default: "2".
headertemplate	(Optional) Id – Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
headerpick	(Optional) Id – Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
buytemplate	(Optional) Id – Specifies the unique id of the template to be shown in the lower right corner of the choose form for controlling the details of a purchase transaction. If empty, no buy template is utilized. Default: Empty.
xactspecial	(Optional) Integer – When a buy template is shared between two or more portals or things, the template behavior may need to be tailored based on the usage. If this need arises, this attribute specifies a unique value that identifies this particular usage. By assigning a different value to each usage and keying on it within the template's Position script, you can tailor the template appropriately. Default: "0".
selltemplate	(Optional) Id – Specifies the unique id of the template to be shown when the user attempts to delete an item. This allows you to enable the selling of items for money. If empty, no sell template is utilized. Default: Empty.
candidatepick	(Optional) Id – Specifies the unique id of a pick that will contain a dynamically generated Candidate tag expression for use in determining the list of available objects to choose from. If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
candidatefield	(Optional) Id – Specifies the unique id of a text-based field that contains the Candidate tag expression used to determine the list of available objects to choose from. This field must exist within the pick identified by the "candidatepick" attribute (above). If empty, the "candidate" child element defines the tag expression to use. Default: Empty.
prereqtarget	(Optional) Set – Designates the container against which all pre-requisite tests need to be performed when determining the list of items available for selection. When displacement is utilized, pre-requisites need to be tested against the container to which the new picks will ultimately be added. Must be one of these values: ▪ container – The default parent container is used. ▪ parent – The next parent up the hierarchy is used, which parallels the corresponding displacement target. ▪ hero – The top-level hero is used, which parallels the corresponding displacement target. ▪ Default: "container".

empty	(Optional) Text – Specifies the text message to be displayed if the user attempts to select an option and there are no available items to choose from. If empty, a default message is displayed. Default: Empty.
showfrozenfixed	(Optional) Boolean – Indicates whether the table must be converted to a "fixed" table whenever the table is designated as frozen. Default: "no".
showfixedlast	(Optional) Boolean – Indicates whether all non-deletable picks within the table are sorted to the end of the list of picks shown. Default: "no".
allowuserorder	(Optional) Boolean – Indicates whether the items in the table can be re-ordered by the user. If enabled, the specified component must designate a suitable ordering field or a separate component with such a field must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
ordercomponent	(Optional) Id – Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
linkage	(Optional) Id – Specifies the unique id of a thing that will be used as a linkage. When a new pick is added to the table, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
alwaysupdate	(Optional) Boolean – Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_dynamic" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

list	An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
candidate	An optional "candidate" element may appear as defined by the given link. This element defines a Candidate Tag Expression for the portal.
restriction	An optional "restriction" element may appear as defined by the given link. This element defines a Restriction Tag Expression for the portal that identifies things which cannot be selected if they've already been added to this table.
needtag	Zero or more "needtag" elements may appear as defined by the given link. This element defines a tag relationship that must exist between a prospective object and the container in order to list the object among the available items.
denytag	Zero or more "denytag" elements may appear as defined by the given link. This element defines a tag relationship that must **not** exist between a prospective object and the container in order to list the object among the available items.
xacttag	Zero or more "xacttag" elements may appear as defined by the given link. This element defines a tag that is assigned to the transaction pick while the choose form is visible.
secondary	An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.
existence	An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.
autotag	Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to each added thing.
chosen	An optional "chosen" element may appear as defined by the given link. This element defines a Chosen Script for the portal.
titlebar	An optional "titlebar" element may appear as defined by the given link. This element defines a TitleBar Script for the portal.
description	An optional "description" element may appear as defined by the given link. This element defines a Description Script for the portal.
headertitle	An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle

	Script for the portal.
additem	An optional "additem" element may appear as defined by the given link. This element defines a AddItem Script for the portal.

The "list" Element

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the List tag expression.

The "candidate" Element

The "candidate" element defines a Candidate Tag Expression for the portal that limits the set of things/picks that are available for selection. If this element is omitted entirely, then the items available for selection must satisfy the List tag expression instead (above). The complete list of attributes for this element is below.

inheritlist	(Optional) Boolean – Indicates whether the List tag expression (above) is automatically inherited into the Candidate tag expression. If inherited, all available objects must satisfy both the Candidate tag expressions and the List tag expression. This eliminates the need to redundantly maintain the same filter logic within both tag expressions. If not inherited, then the Candidate tag expression supersedes the List tag expression. Default: "no".
PCDATA	TagExpr – Specifies the code comprising the Candidate tag expression.

The "restriction" Element

The "restriction" element defines a Restriction Tag Expression for the portal that further limits the set of things/picks that are available for selection. This tag expression is compared against all picks that have already been added to **this table**. Any object that already exists within the table is precluded from being selected again, resulting in it being omitted from the list of available choices. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Restriction tag expression.

The "needtag" Element

The "needtag" element defines a tag relationship that must exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it has at least one matching tag with the same id in a separate tag group. If the tag is not found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

container	Id – Specifies the unique id of the tag group to utilize within the container.
thing	Id – Specifies the unique id of the tag group to check within the thing/pick.
usehero	(Optional) Boolean – Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

The "denytag" Element

The "denytag" element defines a tag relationship that must **not** exist between the object to be added and the prospective container. Tags from one tag group are enumerated within the container, then the object is tested to make sure that it does not possess any matching tags with the same ids in a separate tag group. If any matching tags are found, the object is not valid for selection and omitted from the available list. The complete list of attributes for this element is below.

container	Id – Specifies the unique id of the tag group to utilize within the container.
thing	Id – Specifies the unique id of the tag group to check within the thing/pick.
usehero	(Optional) Boolean – Indicates whether the container tags are pulled from the prospective container for the new pick or the hero. This distinction can be important when using displacement. Default: "no".

The "xacttag" Element

The "xacttag" element specifies a tag that is automatically added to the transaction pick while the choose form is shown. These tags allow you to indicate contextual information about where the buy template is being used so that you can tailor the behavior appropriately. The complete list of attributes for this element is below.

tag	Id – Specifies the unique id of the tag to define within the tag group "transact".

The "secondary" Element

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "existence" Element

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "chosen" Element

The "chosen" element defines a Chosen Script for the portal, which synthesizes the text to be displayed as the chosen item within the portal. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Chosen script.

The "titlebar" Element

The "titlebar" element defines a TitleBar Script for the portal, which synthesizes the text to be displayed at the top of the choose form. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the TitleBar script.

The "description" Element

The "description" element defines a Description Script for the portal, which synthesizes the text to be displayed within the description region of the choose form for the currently selected item on the left. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Description script.

The "headertitle" Element

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the HeaderTitle script.

The "additem" Element

The "additem" element defines a AddItem Script for the portal that synthesizes the text to be displayed within the "add" item at the bottom of the table, where the user will click to add a new item. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the AddItem script.

Example

The following example demonstrates what a dynamic table portal might look like. All default values are assumed for optional attributes.

```
<portal id="arMelee" style="tblNormal">
  <table_dynamic component="Gear"
      showtemplate="arWpnPick" choosetemplate="arWpnThing"
      buytemplate="BuyCash" selltemplate="SellCash">
    <list>component.WeapMelee</list>
    <candidate>!Equipment.Natural</candidate>
    <description/>
    <headertitle>
      @text = "Melee Weapons"
      </headertitle>
    <additem>
      @text = "Add New Melee Weapons"
      </additem>
    </table_dynamic>
  </portal>
```

Category: Kit Reference

TableAuto Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

Contents

The "table_auto" Element

Auto tables are a special kind of dynamic table that can only ever contain a single item. As such, an auto table looks like a dynamic table until the user clicks on the "add" item at the bottom. Instead of displaying a choose form, a new instance of a specific item is added to the table. This is extremely useful for tables of journal entries and character portraits. Each auto table is defined via the use of the "table_auto" element. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component that all shown objects must be derived from.
autothing	Id – Specifies the unique id of the thing to be added to the table whenever the user clicks on the "add" item at the bottom.
showtemplate	Id – Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
showsortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
showgaphorz	(Optional) Integer – Specifies the gap along the horizontal axis to insert between items that exist within the table. Default: "0".
showgapvert	(Optional) Integer – Specifies the gap along the vertical axis to insert between items that exist within the table. Default: "0".
columns	(Optional) Integer – Specifies the number of columns of data to display within the table. Default: "1".
scrollable	(Optional) Boolean – Indicates whether the table contents can be scrolled by the user. By default, a scroller is shown whenever the number of items exceeds the visible space, but you can disable this behavior. Default: "yes".
addtemplate	(Optional) Id – Specifies the unique id of the template to be used for the "add" item that always appears at the bottom of the table and that users will click on to add an item to the table. This allows detailed controlled when the simple "additem" script is not sufficient. If empty, the "additem" element must be specified. Default: Empty.
addpick	(Optional) Id – Specifies the unique id of a thing that is associated with the "add" item at the bottom of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "addtemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
addspace	(Optional) Integer – Specifies the additional vertical space to be inserted when displaying the simple "add" item at the bottom of the table by using the "additem" script. The height of the item is based on the font height of the text shown, so this attribute allows you to insert additional padding if you wish. Default: "2".
headertemplate	(Optional) Id – Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
headerpick	(Optional) Id – Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.
showfrozenfixed	(Optional) Boolean – Indicates whether the table must be converted to a "fixed" table whenever the table is designated as frozen. Default: "no".
	(Optional) Boolean – Indicates whether the items in the table can be re-ordered by the user. If enabled, the

allowuserorder	specified component must designate a suitable ordering field or a separate component with such a field must be specified via the "ordercomponent" attribute. Default: "no". **NOTE!** Verify that whatever sort set you use for showing the items includes the designated ordering field as its first sort key.
ordercomponent	(Optional) Id – Specifies the unique id of an alternate component that possesses a suitable ordering field. This attribute is only applicable when the table supports user ordering. If empty, the ordering field is dictated by the component associated with the table. Default: Empty.
linkage	(Optional) Id – Specifies the unique id of a thing that will be used as a linkage. When a new pick is added to the table, that pick has an automatic linkage setup to any existing pick derived from the specified thing. If no derived pick exists when the new pick is added, no linkage is ever created. If empty, no linkage is established. Default: Empty.
alwaysupdate	(Optional) Boolean – Indicates whether the table must be dynamically updated after any modification to the actor so that the influence of other changes are always visually reflected to the user. Default: "no". **NOTE!** This option can significantly slow down display updates on slower computers, so only enable this if truly necessary.

The "table_auto" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

list	An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
secondary	An optional "secondary" element may appear as defined by the given link. This element defines a Secondary Tag Expression that is associated with every new pick added via the portal.
existence	An optional "existence" element may appear as defined by the given link. This element defines an Existence Tag Expression that is associated with every new pick added via the portal.
autotag	Zero or more "autotag" elements may appear as defined by the given link. This element specifies tags that are automatically assigned to each added thing.
headertitle	An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle Script for the portal.
additem	An optional "additem" element may appear as defined by the given link. This element defines a AddItem Script for the portal.

The "list" Element

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. Regardless of this tag expression, all picks added via this portal are always shown within it, enabling deletion of any object added through the table. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the List tag expression.

The "secondary" Element

The "secondary" element defines a Secondary Tag Expression that is automatically associated with every new pick added via the portal. This new tag expression is treated like an additional Container Tag Expression for the pick that must also be satisfied. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "existence" Element

The "existence" element defines an Existence Tag Expression that is automatically associated with every new pick added via the portal. If a pick ever fails to satisfy the tag expression during an evaluation cycle, the pick is automatically deleted. The complete list of attributes for this element is below.

phase	(Optional) Id – Specifies the unique id of the evaluation phase during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
priority	Integer – Specifies the evaluation priority during which the tag expression is tested. If empty, the default timing is used from the definition file. Default: Empty.
PCDATA	TagExpr – Specifies the code comprising the Secondary tag expression.

The "headertitle" Element

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the HeaderTitle script.

The "additem" Element

The "additem" element defines a AddItem Script for the portal that synthesizes the text to be displayed within the "add" item at the bottom of the table, where the user will click to add a new item. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the AddItem script.

Example

The following example demonstrates what an auto table portal might look like. All default values are assumed for optional attributes.

```
<portal id="peImages" style="tblNormal">
  <table_auto component="UserImage"
      showtemplate="peImage" autothing="mscUserImg">
    <headertitle>
      @text = "Gallery"
      </headertitle>
    <additem>
      @text = "Add Another Image"
      </additem>
    </table_auto>
  </portal>
```

Category: Kit Reference

SettingEdit Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "setting_edit" Element

The "setting_edit" element is used in a single location within HL. This portal corresponds to the button that allows the user to edit the various Settings on the Configure Hero form. It's purpose is to allow you to position the button where you want it on the Configure Hero form. There are no attributes or child elements for this element.

Example

The following example demonstrates what a settingedit portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfEdit" style="special"
      tiptext="Click here to change the settings governing your character.">
  <setting_edit/>
  </portal>
```

Category: Kit Reference

SettingSummary Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "setting_summary" Element

The "setting_summary" element is used in a single location within HL. This portal corresponds to the table that displays the currently selected Settings on the Configure Hero form. It's purpose is to allow you to position and size the table on the Configure Hero form. There are no attributes or child elements for this element.

Example

The following example demonstrates what a setting_summary portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfSummary" style="special">
  <setting_summary/>
  </portal>
```

Category: Kit Reference

Alliance Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "alliance" Element

The "alliance" element is used in a single location within HL. This portal corresponds to the menu that allows the user to toggle whether an actor is an ally or enemy on the Configure Hero form. It's purpose is to allow you to position and size the menu on the Configure Hero form. There are no attributes or child elements for this element.

Example

The following example demonstrates what a alliance portal might look like. All default values are assumed for optional attributes.

```
<portal id="cnfAlly" style="special">
  <alliance/>
  </portal>
```

Category: Kit Reference

OutputLabel Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "output_label" Element

The role of the "output_label" element is identical to the "label" element, except that it is designed for use exclusively within character sheet output. Any data that you want to display within a sheet (i.e. most everything within a sheet) will require the use of an "output_label" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the label contents may be employed within a given output label portal. That means you may use **either** the "text" attribute, the "field" attribute, **or** the "labeltext" script to define the contents. Use of multiple mechanisms will result in a compilation error.

text	(Optional) Text – Specifies a string of literal text to be displayed within the label. Default: Empty.
field	(Optional) Id – Specifies the unique id of the field whose value is to be displayed within the label. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based label is not allowed. Default: Empty.
ismultiline	(Optional) Boolean – Indicates whether the label text is to be treated as multi-line or merely a single line of output. Default: "no".

The "output_label" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

labeltext	An optional "labeltext" element may appear as defined by the given link. This element defines a Label Script that is used for synthesizing the text to be output.

The "labeltext" Element

The "labeltext" element defines a Label Script for the portal. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Label script.

Example

The following example demonstrates what an output label portal might look like. All default values are assumed for optional attributes.

```
<portal id="name" style="outNameLg">
  <output_label field="name"/>
  </portal>

<portal id="oHeroName" style="outHeroNam">
  <output_label>
    <labeltext>
      @text = hero.actorname
      var result as number
      result = compare(@text, "")
      if (result = 0) then
        @text = "- Unnamed Hero -"
        endif
      </labeltext>
    </output_label>
  </portal>
```

Category: Kit Reference

OutputImage Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "image_field" Element

The role of the "output_image" element is an amalgam of the various "image" elements, except that it is designed for use exclusively within character sheet output. Any images that you want to display within a sheet will require the use of an "output_image" element. The complete list of attributes for this element is below.

IMPORTANT! Only one mechanism for specifying the image contents may be employed within a given output image portal. That means you may use **either** the "image" attribute **or** the "field" attribute to define the contents. Use of multiple mechanisms will result in a compilation error.

image	(Optional) Text – Specifies the filename of a bitmap image within the data file folder for the game system. The given image is displayed within the portal.
field	(Optional) Id – Specifies the unique id of the field whose contents dictate the image to to be displayed within the portal. The field may contain a user-defined image, a reference image, or a filename of a bitmap image within the data file folder for the game system. The field must exist within the pick/thing associated with the containing template. If this portal is not defined within a template, a field-based image is not allowed.
istransparent	(Optional) Boolean – Indicates whether the image should be treated as transparent, wherein the pixel color at position 0,0 is considered transparent throughout the image. Default: "no".

Example

The following example demonstrates what an output image portal might look like. All default values are assumed for optional attributes.

```
<portal id="oHLLogo" style="outNormal">
  <output_image image="sheet_hllogo.bmp"/>
  </portal>

<portal id="image" style="outNormal">
  <output_image field="acTacImage"/>
  </portal>
```

Category: Kit Reference

OutputTable Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

Contents

The "output_table" Element

Output tables are exclusively for use within character sheet output, hence the name. Due to their behavior, output tables are very similar to fixed tables, with the primary distinction being that output tables can support variable height items. Each output table is defined via the use of the "output_table" element. The complete list of attributes for this element is below.

component	Id – Specifies the unique id of the component that all shown objects must be derived from.
showtemplate	Id – Specifies the unique id of the template to be used for displaying the picks that have been added to the table.
showpicks	(Optional) Set – Designates the source from which the picks shown are retrieved from. Must be one of these values: ▪ container – The picks shown are from the implicitly identified container. If the containing scene is a form associated with a gizmo, the gizmo is used, else the actor is used. ▪ hero – The picks shown are from the active actor. ▪ actor – The picks shown represent all actors in the entire portfolio. ▪ lead – The picks shown represent all lead actors in the entire portfolio. ▪ minion – The picks shown are all immediate minions for the active actor. ▪ Default: "container".
showsortset	(Optional) Id – Specifies the unique id of the sort set to be used for sequencing the items that exist within the table. If empty, all objects are sorted by name. Default: Empty.
showgaphorz	(Optional) Integer – Specifies the gap along the horizontal axis to insert between items that exist within the table. Default: "0".
showgapvert	(Optional) Integer – Specifies the gap along the vertical axis to insert between items that exist within the table. Default: "0".
columns	(Optional) Integer – Specifies the number of columns of data to display within the table. Default: "1".
varyheight	(Optional) Boolean – Indicates whether the items output within the table can be of varying height. This is extremely valuable for material like special abilities, journal entries, etc. Such material can have description text that ranges from short to long, and it's necessary that each item in the table have a suitable height that matches its contents. Default: "no".
headertemplate	(Optional) Id – Specifies the unique id of the template to be used for a header item that appears at the top of the table. This allows you to add column headers above various pieces of information in the table. If empty, the "headertitle" element dictates whether a header is displayed above the table. Default: Empty.
headerpick	(Optional) Id – Specifies the unique id of a thing that is associated with the header item at the top of the table. HL will retrieve any pick based on this thing that exists within the target container and use it. This allows you to control what fields can be accessed from the "headertemplate". If empty, the "actor" pick is used, except within a gizmo, where its "defaultthing" is used instead. Default: Empty.

The "output_table" element also possesses child elements that define additional behaviors of the portal. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

list	An optional "list" element may appear as defined by the given link. This element defines a List Tag Expression for the portal.
headertitle	An optional "headertitle" element may appear as defined by the given link. This element defines a HeaderTitle

Script for the portal.

The "list" Element

The "list" element defines a List Tag Expression for the portal that limits the set of picks that are shown. The complete list of attributes for this element is below.

PCDATA TagExpr – Specifies the code comprising the List tag expression.

The "headertitle" Element

The "headertitle" element defines a HeaderTitle Script for the portal that synthesizes the text to be displayed at the top of the table as a header. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the HeaderTitle script.

Example

The following example demonstrates what an output table portal might look like. All default values are assumed for optional attributes.

```
<portal id="oJournTbl" style="outNormal">
  <output_table component="Journal" varyheight="yes"
      showtemplate="oJrnPick" showpicks="yes">
    </output_table>
  </portal>

<portal id="oAdjust" style="outNormal">
  <output_table component="Adjustment" showtemplate="oAdjPick" columns="2">
    <list>Helper.Activated</list>
    <headertitle>
      @text = "Activated Adjustments"
      </headertitle>
    </output_table>
  </portal>
```

Category: Kit Reference

OutputDots Element (Data)

Context: HL Kit … Kit Reference … Data File Reference … Portal Element (Data)

The "output_dots" Element

The role of the "output_dots" element is a special purpose "output_label" portal that it is designed for use exclusively within character sheet output. This portal makes it possible to easily insert a series of alternating dots and spaces between two portals within character sheet output. The dots are guaranteed to be vertically aligned, regardless of the span over which they cover, ensuring that a sequence of items in a table that use the dot spanning will look clean and consistent. This portal behaves just like an output_label portal in other respects, so be sure to assign a style the provides the font in which to render the dot sequence. There are no attributes or child elements for this portal.

Example

The following example demonstrates what an output dots portal might look like. All default values are assumed for optional attributes.

```
<portal id="dots" style="outPlain">
  <output_dots/>
  </portal>
```

Category: Kit Reference

OutputSeparator Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference ... Portal Element (Data)

The "outtput_separator" Element

The role of the "output_separator" element is a special separator portal for use within sheet output. The portal inserts a vertical or horizontal bar of solid black between groupings of portals, acting as a visual separator between them. The complete list of attributes for this element is below.

isvertical	(Optional) Boolean – Indicates whether the separator should be drawn horizontally or vertically. Default: "no".

Example

The following example demonstrates what an output separator portal might look like. All default values are assumed for optional attributes.

```
<portal id="separator" style="oSeparator">
  <output_separator isvertical="no"/>
  </portal>
```

Category: Kit Reference

Template Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

Contents

The "template" Element

Templates are collection of related portals that are all linked to a common thing or pick. Each template is specified through the use of a "template" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the template. This id is used in all references to the template.
name	Text – Public name associated with the template. Maximum length of 50 characters.
compset	Id – Specifies the unique id of the component set that things/picks shown within the template derive from.
width	(Optional) Integer – Specifies the initial width to be utilized for the template. If empty, the width is automatically determined by a set of rules. Default: Empty.
height	(Optional) Integer – Specifies the initial height to be utilized for the template. Default: "100".
marginhorz	(Optional) Integer – Specifies the margin gap included at both ends along the horizontal axis. The usable width of the template is the actual width minus double the horizontal margin. Default: "0".
marginvert	(Optional) Integer – Specifies the margin gap included at both ends along the vertical axis. The usable height of the template is the actual height minus double the vertical margin. Default: "0".
istransaction	(Optional) Boolean – Indicates whether this template is intended for use with buy or sell transactions. Default: "no".

The "template" element also possesses child elements that define various facets of the template. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the template is shown to the user.
portal	One or more "portal" elements must appear as defined by the given link. This element specifies the individual portals which exist within the template.
position	An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the portals are sized and positioned within the template.
header	An optional "header" element may appear as defined by the given link. This element defines a Header Script that coordinates the sizing and positioning of portals used within the header for the template.

The "live" Element

The "live" element defines a Live Tag Expression for the template, which determines whether the template is shown. The tag expression is compared against the tags assigned to the container associated with the template (e.g. the actor). If the tag expression is satisfied, the template is visible, else it is hidden. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "position" Element

The "position" element defines a Position Script for the template, which performs all the appropriate sizing and positioning of the contained portals within the region of the template. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Position script.

The "header" Element

The "header" element defines a Header Script for the template, which works exactly like the Position script, except for the portals it manipulates. This script is only used if the template is used as a dual header and contains specifically designated header portals. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the Header script.

Example

The following example demonstrates what a "template" element might look like. All default values are assumed for optional attributes.

```
<template id="baAttrPick" name="Attribute Pick" compset="Attribute"
    marginhorz="18" marginvert="9">
  <portal id="name" style="lblXLarge" showinvalid="yes">
    <label field="name"/>
    </portal>
  <portal id="value" style="incrSimple">
    <incrementer field="trtUser"/>
    <mouseinfo mousepos="middle+above">
      @text = "Adjust attribute by clicking on the arrows."
      </mouseinfo>
    </portal>
  <portal id="info" style="actInfo">
    <action action="info"/>
    </portal>
  <position><![CDATA[
    ~set up our height based on our tallest portal
    height = portal[info].height

    ~if this is a "sizing" calculation, we're done
    if (issizing <> 0) then
      done
      endif

    ~position our tallest portal at the top
    portal[info].top = 0

    ~center the other portals vertically
    perform portal[name].centervert
    perform portal[value].centervert

    ~position the info portal on the far right
    perform portal[info].alignedge[right,0]

    ~position the incrementer to the left of the info portal (plus a gap)
    perform portal[value].alignrel[rtol,info,-10]

    ~position the name on the left and limit its width to available space
    portal[name].left = 0
    portal[name].width = minimum(portal[name].width,portal[value].left - portal[name].left - 10)
    </position>
  </template>
```

Category: Kit Reference

Layout Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference

Contents

The "layout" Element

Layouts contain an assortment of portals and templates that serve a unified purpose when presenting information to the user. Each layout is specified through the use of a "layout" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the layout. This id is used in all references to the layout.
marginhorz	(Optional) Integer – Specifies the margin gap included at both ends along the horizontal axis. The usable width of the layout is the actual width minus double the horizontal margin. Default: "0".
marginvert	(Optional) Integer – Specifies the margin gap included at both ends along the vertical axis. The usable height of the layout is the actual height minus double the vertical margin. Default: "0".

The "layout" element also possesses child elements that define various facets of the layout. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the template is shown to the user.
portalref	Zero or more "portalref" elements may appear as defined by the given link. This element specifies the individual portals which are utilized within the layout.
templateref	Zero or more "templateref" elements may appear as defined by the given link. This element specifies the individual templates which are utilized within the layout.
position	An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the portals are sized and positioned within the template.

The "live" Element

The "live" element defines a Live Tag Expression for the layout, which determines whether the layout is shown. The tag expression is compared against the tags assigned to the container associated with the layout (e.g. the actor). If the tag expression is satisfied, the layout is visible, else it is hidden. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "portalref" Element

The "portalref" element identifies a portal used within the layout and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

portal	Id – Unique id of the portal to be utilized within the layout.
reference	(Optional) Id – Specifies the alternate unique id to be used when referencing this portal within the Position script. This makes it possible to re-use the same portal more than once within the same layout, giving each instance a different logical id. If empty, the reference id is simply the portal id. Default: Empty.
taborder	(Optional) Integer – Specifies the relative tab order position of the portal and its contents within the overall layout. As long as the tab orders are different for all portals and templates within the layout, HL can properly sequence the flow of control. Default: "0".

The "templateref" Element

The "templateref" element identifies a template used within the layout and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

template	Id – Unique id of the template to be utilized within the layout.
reference	(Optional) Id – Specifies the alternate unique id to be used when referencing this template within the Position script. This makes it possible to re-use the same template more than once within the same layout, giving each instance a different logical id. If empty, the reference id is simply the template id. Default: Empty.
thing	(Optional) Id – Specifies the unique id of the thing or pick that is associated with this template. The given object is used with the template when determining the contents of fields and such. If empty, no thing/pick is associated with the template, which means that the template cannot contain any portals that utilize a field reference. Default: Empty.
ispick	(Optional) Boolean – Indicates whether the template is associated with a pick or a thing. If a pick, the first pick within the container that based on the specified thing is used. Default: "yes".
taborder	(Optional) Integer – Specifies the relative tab order position of the template and its contents within the overall layout. As long as the tab orders are different for all portals and templates within the layout, HL can properly sequence the flow of control. Default: "0".

The "position" Element

The "position" element defines a Position Script for the layout, which performs all the appropriate sizing and positioning of the contained portals and templates within the region of the layout. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Position script.

Example

The following example demonstrates what a "layout" element might look like. All default values are assumed for optional attributes.

```
<layout id="journal">
  <portalref portal="jrTitle"/>
  <portalref portal="journal" taborder="10"/>
  <templateref template="jrHeader" thing="actor"/>
  <position>
    ~configure whether our header is visible or not
    ~Note: If the header is non-visible, no space is allotted for it below.
    template[jrHeader].visible = 1

    ~position the title at the top, followed by the template, and then the table
    perform portal[jrTitle].autoplace
    perform template[jrHeader].autoplace[6]
    perform portal[journal].autoplace[9]
    </position>
  </layout>
```

Category: Kit Reference

Panel Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

Contents

The "panel" Element

Panels contain one or more layouts and orchestrate the presentation of material for a specific region within HL. Panels are used to define the contents of the various tabs shown across the top for editing character data, as well as for summary panels shown on the right. Each panel is specified through the use of a "panel" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the panel. This id is used in all references to the panel.
name	Text – Public name associated with the panel. Maximum length of 50 characters.
order	(Optional) Integer – Specifies the sequence in which this panel is displayed to the user. For a tab-based panel, this value controls the order in which the tabs are shown across the top. For a summary panel, this value controls the default order in which the summary panels are shown. Default: "0".
issummary	(Optional) Boolean – Indicates whether this panel is intended for use as a summary panel or a tab-based panel. Default: "no".
marginhorz	(Optional) Integer – Specifies the margin gap included at both ends along the horizontal axis. The usable width of the panel is the actual width minus double the horizontal margin. Default: "0".
marginvert	(Optional) Integer – Specifies the margin gap included at both ends along the vertical axis. The usable height of the panel is the actual height minus double the vertical margin. Default: "0".
underlay	(Optional) Text – Specifies the filename of a bitmap image that is blitted on top of the background and beneath all other visual elements within the panel. This bitmap file is assumed to reside within the data file folder for the game system. If empty, no underlay bitmap is utilized. Default: Empty.

The "panel" element also possesses child elements that define various facets of the panel. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the panel is shown to the user.
layoutref	Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the panel.
position	An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the panel.

The "live" Element

The "live" element defines a Live Tag Expression for the panel, which determines whether the panel is shown. The tag expression is compared against the tags assigned to the container associated with the panel (e.g. the actor). If the tag expression is satisfied, the panel is visible, else it is hidden. For tab panels, hidden implies that the tab itself does not appear. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "layoutref" Element

The "layoutref" element identifies a layout used within the panel and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

layout	Id – Unique id of the layout to be utilized within the panel.
reference	(Optional) Id – Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same panel, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

The "position" Element

The "position" element defines a Position Script for the panel, which performs all the appropriate sizing and positioning of the contained layouts within the panel. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Position script.

Example

The following example demonstrates what a "panel" element might look like. All default values are assumed for optional attributes.

```
<panel id="basics" name="Basics" order="110"
    marginhorz="5" marginvert="5">
  <live>!HideTab.basics</live>
  <layoutref layout="basics"/>
  <position>
    ~script code goes here
    </position>
  </panel>
```

Category: Kit Reference

Form Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference

Contents

The "form" Element

Forms contain one or more layouts and orchestrate the presentation of material within a standalone form within HL. Forms are used to define the contents of the Configure Hero form, the Tactical Console, and other visual pieces. Each form is specified through the use of a "form" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the form. This id is used in all references to the form.
name	Text – Public name associated with the form. Maximum length of 50 characters.
showbutton	(Optional) Boolean – Indicates whether a button should be added at the bottom of the form that allows the user to close the form. Default: "yes".
entity	(Optional) Id – Specifies the unique of an entity for which this form is designed to edit the contents. If empty, the form cannot be used for editing any gizmos. Default: Empty.
marginhorz	(Optional) Integer – Specifies the margin gap included at both ends along the horizontal axis. The usable width of the panel is the actual width minus double the horizontal margin. Default: "0".
marginvert	(Optional) Integer – Specifies the margin gap included at both ends along the vertical axis. The usable height of the panel is the actual height minus double the vertical margin. Default: "0".
defwidth	(Optional) Integer – Specifies the default initial width to be used for the form. Default: "300".
defheight	(Optional) Integer – Specifies the default initial height to be used for the form. Default: "300".
minwidth	(Optional) Integer – Specifies the minimum width to be enforced for the form. Default: "0".
minheight	(Optional) Integer – Specifies the minimum height to be enforced for the form. Default: "0".
maxwidth	(Optional) Integer – Specifies the maximum width to be enforced for the form. Default: "0".
maxheight	(Optional) Integer – Specifies the maximum height to be enforced for the form. Default: "0".

The "form" element also possesses child elements that define various facets of the form. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

layoutref	Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the form.
position	An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the form.

The "layoutref" Element

The "layoutref" element identifies a layout used within the form and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

layout	Id – Unique id of the layout to be utilized within the form.
reference	(Optional) Id – Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same form, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

The "position" Element

The "position" element defines a Position Script for the form, which performs all the appropriate sizing and positioning of the contained layouts within the form. The complete list of attributes for this element is below.

PCDATA Script – Specifies the code comprising the Position script.

Example

The following example demonstrates what a "form" element might look like. All default values are assumed for optional attributes.

```
<panel id="config" name="Configure">
  <layoutref layout="configure"/>
  <position>
    ~render the layout to generate its dimensions
    perform layout[configure].render
    ~set the width and height of the form to the dimensions of the layout
    width = layout[configure].width
    height = layout[configure].height
    </position>
  </panel>
```

Category: Kit Reference

Sheet Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

Contents

The "sheet" Element

Sheets contain one or more layouts and orchestrate the presentation of material within a specific page of a character sheet. Each sheet is specified through the use of a "sheet" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the sheet. This id is used in all references to the sheet.
name	Text – Public name associated with the sheet. Maximum length of 50 characters.
landscape	(Optional) Boolean – Indicates whether the sheet is to be output in landscape or portrait orientation. Default: "no".
spillover	(Optional) Boolean – Indicates whether this sheet utilized spillover behaviors or not. If spillover behavior is enabled, the sheet is output repeatedly until all objects rendered via the page have been output. Default: "no".

The "sheet" element also possesses child elements that define various facets of the sheet. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the sheet is actually output for the character.
layoutref	Zero or more "layoutref" elements may appear as defined by the given link. This element specifies the individual layouts which are utilized within the sheet.
position	An optional "position" element may appear as defined by the given link. This element defines a Position Script through which the contained layouts are sized and positioned within the sheet.

The "live" Element

The "live" element defines a Live Tag Expression for the sheet, which determines whether the sheet is included in the output. The tag expression is compared against the tags assigned to the actor being output. If the tag expression is satisfied, the sheet is output, else it is omitted. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "layoutref" Element

The "layoutref" element identifies a layout used within the sheet and assigns it a logical name for use within the Position script. The complete list of attributes for this element is below.

layout	Id – Unique id of the layout to be utilized within the sheet.
reference	(Optional) Id – Specifies the alternate unique id to be used when referencing this layout within the Position script. This makes it possible to re-use the same layout more than once within the same sheet, giving each instance a different logical id. If empty, the reference id is simply the layout id. Default: Empty.

The "position" Element

The "position" element defines a Position Script for the sheet, which performs all the appropriate sizing and positioning of the contained layouts within the sheet. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Position script.

Example

The following example demonstrates what a "sheet" element might look like. All default values are assumed for optional attributes.

```
<sheet id="standard2" name="Standard character sheet, page #2" spillover="yes">
  <layoutref layout="oStandard2" reference="left"/>
  <layoutref layout="oStandard2" reference="right"/>
  <position>
    ~setup the gap to be used between the various sections of the character sheet
    autogap = 40
    global[sectiongap] = autogap

    ~calculate the width of the two columns of the character sheet, leaving a
    ~suitable center gap between them
    var colwidth as number
    colwidth = (width - 50) / 2

    ~output the layout on the lefthand side with whatever information will fit
    layout[left].width = colwidth
    layout[left].height = height
    perform layout[left].render

    ~output the layout on the righthand side with whatever information will fit
    layout[right].width = colwidth
    layout[right].height = height
    perform layout[right].render
    </position>
  </sheet>
```

Category: Kit Reference

Dossier Element (Data)

Context: HL Kit … Kit Reference … Data File Reference

Contents

The "dossier" Element

Dossiers represent a logical set of output for a portfolio, such as a character sheet, statblock, or even export data for use with another product. Each dossier is specified through the use of a "dossier" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the dossier. This id is used in all references to the dossier. When the dossier is being rendered, a global tag "dossier.<this id>" is added to the portfolio allow you to see which dossier is being rendered.
name	Text – Public name associated with the dossier. Maximum length of 100 characters.

The "dossier" element also possesses child elements that define various facets of the dossier. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

IMPORTANT! Exactly one of these child elements may be specified for each dossier. If multiple are given, a compiler error will be reported. The chosen child element dictates the type of dossier that is being defined and its characteristics.

dossier_sheet	An optional "dossier_sheet" element may appear as defined by the given link. This element defines the structure of a printed dossier, such as a character sheet.
dossier_text	An optional "dossier_text" element may appear as defined by the given link. This element specifies text-based output, such as a statblock.
dossier_export	An optional "dossier_export" element may appear as defined by the given link. This element defines output that is tailored for use within another product, such as d20Pro.

The "dossier_sheet" Element

The "dossier_sheet" element defines the structure of a printed dossier, containing a collection of sheets. The complete list of attributes for this element is below.

grouping	Text – Specifies the name of the group into which this dossier should be placed for display. The groupings are used to logically organize multiple dossiers for presentation to the user, with the various groupings being sorted alphabetically.
default	(Optional) Boolean – Indicates whether this dossier should be the default dossier that is pre-selected for the user when first outputting a printed dossier. Default: "no".

The "dossier_sheet" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the dossier is made available for selection.
sheetref	Zero or more "sheetref" elements may appear as defined by the given link. This element identifies a sheet to be included in the dossier output.

The "live" Element
The "live" element defines a Live Tag Expression for the dossier, which determines whether the dossier is shown to the user as a selection. The tag expression is compared against the tags assigned to the current actor. If the tag expression is satisfied, the dossier can be output, else it is hidden. This is ideal for situations like spellbooks that should only be shown for spellcasters. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "sheetref" Element
The "sheetref" element specifies a sheet to be output as part of the dossier. The sequence in which the sheets are specified dictates the order in which they will be processed in the output. The complete list of attributes for this element is below.

sheet	Id – Specifies the unique id of the sheet to be included in the dossier output.

The "dossier_text" Element

The "dossier_text" element defines the logic used in outputting a text-based dossier, such as a statblock. The complete list of attributes for this element is below.

grouping	Text – Specifies the name of the group into which this dossier should be placed for display. The groupings are used to logically organize multiple dossiers for presentation to the user, with the various groupings being sorted alphabetically.
styles	(Optional) Text – Specifies the output styles that are supported for this dossier. The various output styles you can specify are "plain", "html", and "bbcode". You can combine multiple styles by placing a '+' between them (e.g. "plain+bbcode"). If empty, the "plain" style is assumed. Default: Empty.
default	(Optional) Boolean – Indicates whether this dossier should be the default dossier that is pre-selected for the user when first outputting a text dossier. Default: "no".

The "dossier_text" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

live	An optional "live" element may appear as defined by the given link. This element defines a Live Tag Expression that determines whether the dossier is made available for selection.
synthesize	Zero or more "synthesize" elements may appear as defined by the given link. This element defines the Synthesize Script used to generate the text to be output.

The "live" Element
The "live" element defines a Live Tag Expression for the dossier, which determines whether the dossier is shown to the user as a selection. The tag expression is compared against the tags assigned to the current actor. If the tag expression is satisfied, the dossier can be output, else it is hidden. This is ideal for situations like spellbooks that should only be shown for spellcasters. The complete list of attributes for this element is below.

PCDATA	TagExpr – Specifies the code comprising the Live tag expression.

The "synthesize" Element
The "synthesize" element defines a Synthesize Script for the dossier, which properly generates the text to be output as the dossier. If multiple synthesize scripts are defined, their generated results are appended into one final text stream for output. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Synthesize script.

The "dossier_export" Element

The "dossier_export" element defines the logic used in generating formatted data for use within other products, such as mapping tools and virtual table tops. The complete list of attributes for this element is below.

	(Optional) Text – Specifies the default filename into which the generated data is output. The user is free to specify

filename	any filename he wishes, but this allows HL to automatically name the file correctly for programs with pre-defined import file mechanisms. If empty, the user is required the specify a filename. Default: Empty.
location	(Optional) Text – Specifies the default location (i.e. file system path) in which the generated data is output. As with the "filename" attribute above, you can automatically output the file to a pre-determined location where the target program is expecting to find the file to import. If empty, the user is required to specify the location. Default: Empty.

The "dossier_export" element also possesses child elements that describe its contents. The list of these child elements is below and must appear in the order shown. Click on the link to access the details for each element.

synthesize	Zero or more "synthesize" elements may appear as defined by the given link. This element defines the Synthesize Script used to generate the text to be output.

The "synthesize" Element

The "synthesize" element defines a Synthesize Script for the dossier, which properly generates the text to be output as the dossier. If multiple synthesize scripts are defined, their generated results are appended into one final text stream for output. The complete list of attributes for this element is below.

PCDATA	Script – Specifies the code comprising the Synthesize script.

Example

The following example demonstrates what a "dossier" element might look like. All default values are assumed for optional attributes.

```
<dossier id="standard" name="Standard Character Sheet">
  <dossier_sheet grouping="AtTop" default="yes">
    <sheetref sheet="standard1"/>
    <sheetref sheet="standard2"/>
    </dossier_sheet>
  </dossier>

<dossier id="statblock" name="Character Statblock">
  <dossier_text styles="plain+html+bbcode" grouping="statblock">
    <synthesize>
      ~script code goes here
      </synthesize>
    </dossier_text>
  </dossier>
```

When the "standard" dossier is being rendered, the global tag "dossier.standard" is added to the portfolio. When the "statblock" dossier is being synthesized, the global tag "dossier.statblock" is added to the portfolio.

Category: Kit Reference

Hidden Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference

The "hidden" Element

The "hidden" mechanism comes into play when users want to modify existing data files while keeping their changes separate. This comes in extremely handy if there are certain standard things (e.g. skills, feats, etc.) in your campaign that are forbidden by the GM. The "hidden" mechanism flags an object so that HL treats the thing as if it does not exist, even though it remains technically present. This means a hidden object will never be shown to the user for selection. Each hidden thing is defined through the use of a "hidden" element. The complete list of attributes for this element is below.

type	(Optional) Set – Designates the type of object to be hidden. Must be one of these values: ▪ thing – The object must be defined via a "thing" element. ▪ Default: "thing". **NOTE!** At the present time, only things can be hidden. The design of this mechanism allows it to be extended in the future if there is a need to hide other objects.
id	Id – Specifies the unique id of the object to be hidden. An object of the specified type must exist within this unique id.

NOTE! Since hidden objects are treated as if they don't exist, any references to hidden objects will fail to resolve successfully. This may have nasty implications, because it means that a hidden thing that is bootstrapped by another thing will cause the second thing to fail to compile. Normally, this will simply create a cascading list of a few things that must all be marked as hidden, but sometimes it can cause circular dependency issues. When that occurs, use the "replace" mechanism on things to substitute a thing that is innocuous.

Example

The following example demonstrates what a "hidden" element might look like. All default values are assumed for optional attributes.

```
<hidden type="thing" id="thingid"/>
```

Category: Kit Reference

Edit Thing Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference

The "editthing" Element

The purpose of the "editthing" element is to allow authors to define how new things can be created by users via the integrated Editor within HL. For every type of thing that users are allowed to create, a separate "editthing" element is defined.

[TBD] Details of this mechanism will be documented in the future.

Categories: Kit Reference | TBD - Not Yet Written

FAQ Element (Data)

Context: HL Kit ... Kit Reference ... Data File Reference

The "faq" Element

The data files for every game system provide the means for an author to develop and maintain a list of frequently asked questions (or FAQ). The FAQ serves as an organized repository of notes, explanations, tips, and whatnot for anyone using the data files. The FAQ is not intended as an actual user manual for the data files, but augments the user manual with bits of useful information that don't make sense to include directly within the manual. Each individual FAQ entry is defined through the use of a "faq" element. The complete list of attributes for this element is below.

id	Id – Specifies the unique id of the FAQ entry. This id is used in all references to the FAQ entry.
order	(Optional) Integer – Specifies the order in which this FAQ entry is to appear in the generated FAQ file. All FAQ entries are sorted based on this attribute, with the lower values appearing before higher values. If two or more FAQ entries have the same order value, they are sorted alphabetically based on the topic. This allows you to control the order in which FAQ entries appear, enabling organization of the contents. Default: "100".
topic	Text – Specifies the "title" to displayed for this FAQ entry. A summary of all topics is generated at the top of the FAQ file, allowing users to see all the topics and click on those of interest to get direct access to the details.
PCDATA	Text – Specifies the detailed description text to be displayed for this FAQ entry. The description **may** contain HTML tags, although care should be taken to ensure that the HTML used is very simple, else formatting problems can arise in the generated FAQ file.

Example

The following example demonstrates what a "faq" element might look like. All default values are assumed for optional attributes.

```
<faq id="faqentry" order="1000"
    topic="Add your own FAQ entry here">
    Description of the FAQ entry
  </faq>
```

Category: Kit Reference